Mémoire

Présenté par

Mnacho Echenim

Pour l'obtention de

L'Habilitation à Diriger des Recherches

DES TECHNIQUES D'AUTOMATISATION DE RAISONNEMENTS

Jury :

Eric Gaussier	Professeur Université Grenoble Alpes
Claude Kirchner	Directeur de Recherche INRIA
Silvio Ranise	Directeur de Recherche Fondation Bruno Kessler
Michaël Rusinowitch	Directeur de Recherche INRIA
Viorica Sofronie-Stokkermans	Professeure Université de Koblenz-Landau

9 Novembre 2017

Table des matières

A Sophie, Clément, Anaïs et Elisa

Remerciements

Je tiens avant tout à remercier Viorica Sofronie-Stokkermans, Professeure à l'Université de Koblenz-Landau, Michaël Rusinowitch, Directeur de Recherche à l'INRIA et Silvio Ranise, Directeur de Recherche à la Fondation Bruno Kessler d'avoir accepté la tâche de rapporteur de cette habilitation. J'adresse également mes remerciements chaleureux à Claude Kirchner, Directeur de Recherche à l'INRIA, qui a accepté de participer à ce jury en tant qu'examinateur, et à Eric Gaussier, Professeur à l'Université Grenoble Alpes et Directeur du Laboratoire d'Informatique de Grenoble, qui me fait l'honneur de présider ce jury.

Je remercie tous mes collègues et amis de l'Ensimag et du Laboratoire d'Informatique de Grenoble grâce auxquels j'ai toujours pu travailler dans des conditions agréables, et je remercie tout particulièrement mes collègues de l'équipe CAPP pour leur soutien, et pour tous les échanges, conversations et rires que nous avons pu partager.

Chapitre 1

Introduction

1.1 Présentation

L'automatisation du raisonnement est une idée ancienne, puisque Leibniz imaginait déjà en 1685 qu'il devait être possible de créer une langue universelle et formelle (la *characteristica universalis*) et un algorithme (le *calculus ratiocinator*), qui permettraient de remplacer une fois pour toutes le raisonnement par le calcul et ainsi résoudre les divergences et conflits. L'optimisme suscité par de nombreux progrès dans la formalisation des mathématiques, et caractérisé par l'exclamation de Hilbert en 1930 : "Wir müssen wissen. Wir werden wissen!" (*Nous devons savoir. Nous saurons !*) s'effondra avec les théorèmes d'incomplétude de Gödel en 1931. Mais si le raisonnement ne pourra jamais être entièrement remplacé par le calcul, les contextes dans lesquels il est possible de concevoir des systèmes capables d'effectuer des raisonnements de façon (semi)-automatisée sont de plus en plus nombreux.

Une grande partie des raisonnements effectués en logique propositionnelle sont aujourd'hui complètement automatisés. Les premiers résultats significatifs sur cette logique datent des années 50 avec le programme *Logic Theorist* [137], qui a démontré plus des deux tiers des théorèmes en logique propositionnelle des *Principia Mathematica*. De nombreuses techniques ont ensuite été mises au point pour tester la satisfaisabilité d'une formule propositionnelle de façon efficace. Bien qu'étant décidable, ce problème est NP-complet, mais les outils modernes, basés sur des techniques aussi bien algorithmiques (DPLL [61, 60], CDCL [180]) que d'implémentation (*watched literals* [133]), sont capables de résoudre des problèmes contenant des dizaines de milliers de variables et des millions de clauses en quelques minutes. C'est pourquoi de nombreux problèmes pratiques sont résolus par traduction en des formules propositionnelles et appel à des solveurs SAT ; la logique propositionnelle est encore couramment utilisée en model checking et en reactive system checking. Encore récemment, un problème ouvert vieux de 35 ans, la bicoloration des triplets de Pythagore, a été résolu grâce à un solveur SAT ; la preuve formelle

de ce résultat au format DRAT ayant une taille de plus de 200To [104].

Si plusieurs logiques plus expressives que la logique propositionnelle ont été le thème de nombreux travaux de recherche, comme par exemple les logiques modales, la logique la plus expressive et conservant néanmoins de bonnes propriétés pour l'automatisation est sans doute la logique du premier ordre. Cette logique est suffisamment expressive pour que de nombreuses autres logiques puissent y être encodées, mais le problème de validité [1] y est seulement semi-décidable. Ce résultat est basé sur le théorème de Herbrand qui garantit pour toute formule insatisfaisable l'existence d'un ensemble fini d'instances de cette formule qui est également insatisfaisable. Les premiers démonstrateurs automatiques pour cette logique étaient basés sur une énumération incrémentale des instances de la formule considérée et un test (propositionnel) de la satisfaisabilité de l'ensemble de ces instances [61]. Cette approche a été drastiquement améliorée après l'introduction de la Résolution par Robinson [155]. Ce système d'inférence est constitué d'une unique règle, elle-même basée sur l'unification, et a permis l'implémentation des premiers démonstrateurs automatiques efficaces. De nombreuses techniques mises au point, dont la préférence unitaire et l'ensemble support [178] ou l'algorithme "given clause" [127] ont été intégrés dans Otter [126], le premier démonstrateur automatique capable de résoudre des problèmes non-triviaux efficacement.

Un effort particulier a été consacré au traitement de l'égalité en démonstration automatique. Une façon immédiate de procéder est d'inclure les axiomes de l'égalité à la formule considérée, mais il a rapidement été constaté qu'il était possible d'obtenir des gains de performance significatifs en concevant des outils dans lesquels le symbole de l'égalité est interprété, afin de se passer de ces axiomes. Plusieurs découvertes majeures, dont la règle de Paramodulation [154] et la complétion de Knuth-Bendix [115], qui permet d'orienter des équations, ont mené à la définition du calcul de Superposition [15, 140], qui est utilisé dans une majeure partie des démonstrateurs automatiques modernes les plus efficaces [172, 105, 174, 157]. Ces différentes améliorations ont mené en 1997 à la résolution automatique d'un problème de mathématiques vieux de 50 ans : le système `EQP` a démontré que les algèbres de Robbins sont des algèbres de Boole. Il y a encore de nombreux travaux visant à améliorer les performances des démonstrateurs automatiques, notamment pour gérer l'explosion combinatoire qui peut se produire sur certains problèmes, ou à étendre les capacités de ces démonstrateurs dans l'optique de leur permettre par exemple d'engendrer automatiquement des contre-exemples [50] ou d'effectuer des preuves par induction [42]. De tels travaux permettent d'augmenter la part du raisonnement qui peut être mécanisée.

1. Une formule exprimée dans cette logique est-elle valide ?

1.2 Contributions

Nos travaux de recherche peuvent être classés dans les catégories suivantes.

Solveurs SMT et démonstrateurs automatiques. Les problèmes de satisfaisabilité modulo des théories (SMT) sont des problèmes pouvant être modélisés par des ensembles de clauses fermées dont il faut tester la satisfaisabilité modulo une théorie décidable, comme par exemple l'arithmétique de Presburger ou la théorie des tableaux. Les solveurs SMT sont généralement employés pour résoudre des problèmes avec une structure combinatoire importante, et sont capables de résoudre de tels problèmes bien plus efficacement que les démonstrateurs automatiques génériques les plus efficaces, tels ceux basés sur le calcul de Superposition. Les solveurs SMT nécessitent néanmoins l'implémentation de procédures spécifiques pour chaque théorie pouvant être traitée, tandis que les démonstrateurs automatiques peuvent être employés uniformément pour n'importe quel problème de logique équationnelle du premier ordre. Nous avons étudié comment combiner les points forts de ces deux types d'outils, et étendre les fonctionnalités de solveurs SMT à l'aide de démonstrateurs automatiques. L'objectif de ces travaux est d'obtenir, à partir de solveurs pour des théories de base, des outils capables de résoudre des problèmes dans des théories plus complexes, dans des combinaisons de théories, ou encore de résoudre des problèmes contenant des clauses qui ne sont pas fermées. Ces travaux sont décrits dans le Chapitre 3, et ont été publiés dans [31, 32, 33, 34, 35].

Méthodes d'instanciation. Dans la continuation des travaux précédents, nous avons mis au point des méthodes d'instanciation génériques pouvant être utilisées pour résoudre de nombreux problèmes SMT, en les réduisant à des instances de problèmes dans des théories plus simples. Par exemple, ces méthodes permettent de réduire des problèmes dans la théorie des tableaux avec indices entiers à des problèmes en arithmétique de Presburger. Nous avons également montré comment, partant de méthodes d'instanciation pour certaines théories, construire automatiquement des méthodes d'instanciation dans des combinaisons de plus en plus complexes de ces théories. Les méthodes d'instanciation proposées ont été développées avec un impératif d'efficacité et les formules sont instanciées le moins possible. Ces méthodes sont donc incomplètes en général, et nous avons défini des critères syntaxiques garantissant leur complétude. Nous avons montré que ces méthodes sont complètes pour de nombreuses théories fréquemment utilisées dans les problèmes SMT. Ces travaux sont décrits dans le Chapitre 4, et ont été publiés dans [71, 81, 82, 83].

Raisonnement par induction. La recherche des conséquences inductives

d'une théorie est une tâche complexe, qui est impossible à automatiser dans le cas général. Deux des techniques fréquemment employées pour cette tâche sont l'induction explicite, dans laquelle le schéma d'induction est encodé dans le système d'inférence et l'aide de l'utilisateur peut être requise pour déterminer les invariants adéquats, et l'*induction sans induction*, qui réduit la preuve d'un théorème inductif à une preuve de consistance d'un ensemble bien défini d'axiomes. Nous avons exploré comment intégrer le raisonnement inductif dans différentes procédures de preuve pour la logique du premier ordre. Nous avons ainsi construit un système d'inférence basé sur la méthode des tableaux permettant d'effectuer des raisonnements sur des structures définies par induction structurelle, et étudié l'intégration de ce mode de raisonnement dans des procédures de preuve basées sur la saturation, comme la Résolution ou le calcul de Superposition. Le principe des procédures mises au point est la détection de cycles dans les formules engendrées ; sous certaines conditions, ces cycles exhibent l'occurrence d'un invariant inductif qui, une fois identifié, permet de résoudre le problème considéré de façon standard. Ces travaux sont décrits dans le Chapitre 5, et ont été publiés dans [9, 73].

Raisonnement par abduction. Le but du raisonnement par abduction est d'engendrer des hypothèses permettant d'expliquer des observations imprévues. Une application de ce mode de raisonnement est la vérification automatique de circuits et de programmes, où les hypothèses ont pour but d'expliquer pourquoi un système ne fonctionne pas comme prévu. Si ce mode de raisonnement a été largement étudié en logique propositionnelle à cause de ses nombreuses applications à l'Intelligence Artificielle, il y a beaucoup moins de travaux sur ce thème dans des logiques plus expressives. Nous avons exploré comment adapter des procédures basées sur la saturation pour engendrer les ensembles d'hypothèses recherchées dans le cadre de la logique équationnelle, domaine sur lequel aucune recherche n'avait été menée auparavant. Ces hypothèses sont engendrées en résolvant un problème dual à la recherche d'hypothèses manquantes : la recherche des *impliqués premiers* d'une formule. Nous avons ainsi défini plusieurs calculs qui diffèrent principalement par la façon dont sont traités les symboles pouvant être utilisés pour construire les hypothèses manquantes. Les expérimentations sur les outils développés ont permis de comparer leur efficacité à celle d'autres outils permettant de résoudre des problèmes d'abduction. Ces travaux sont décrits dans le Chapitre 6, et ont été publiés dans [72, 75, 76, 78, 77, 79, 80, 85].

Chapitre 2

Production scientifique

Revues internationales

1. Mnacho Echenim and Nicolas Peltier. A Superposition Calculus for Abductive Reasoning. *Journal of Automated Reasoning*, 2016
2. Vincent Aravantinos, Mnacho Echenim, and Nicolas Peltier. A resolution calculus for first-order schemata. *Fundamenta Informaticae*, 125(2):101–133, 2013
3. Mnacho Echenim and Nicolas Peltier. Instantiation Schemes for Nested Theories. *ACM Transactions on Computational Logic*, 14(2):11, 2013
4. Mnacho Echenim and Nicolas Peltier. An instantiation scheme for satisfiability modulo theories. *Journal of Automated Reasoning*, 48(3), 2012
5. Mnacho Echenim and Nicolas Peltier. Modular instantiation schemes. *Information Processing Letters*, 111(20):989–993, 2011
6. Maria Paola Bonacina and Mnacho Echenim. Theory decision by decomposition. *J. Symb. Comput.*, 45(2):229–260, 2010
7. Maria Paola Bonacina and Mnacho Echenim. On variable-inactivity and polynomial T-satisfiability procedures. *Journal of Logic and Computation*, 18(1):77–96, 2008

Conférences internationales avec comité de lecture

1. Mnacho Echenim and Nicolas Peltier. The binomial pricing model in finance: A formalization in isabelle. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 546–562. Springer, 2017

2. M. Echenim, N. Peltier, and S. Tourret. Quantifier-Free Equational Logic and Prime Implicate Generation. In *CADE 25 (25nd International Conference on Automated Deduction)*. Springer, 2015
3. M. Echenim, N. Peltier, and S. Tourret. A Superposition-Based Approach to Abductive Reasoning in Equational Clausal Logic. In *Proceedings of ADDTC 2014 (Automated Deduction: Decidability, Complexity, Tractability)*, 2014. Invited talk (N. Peltier)
4. M. Echenim, N. Peltier, and S. Tourret. A Deductive-Complete Constrained Superposition Calculus for Ground Flat Equational Clauses. In *4th Workshop on Practical Aspects of Automated Reasoning*, 2014
5. M. Echenim, N. Peltier, and S. Tourret. A Rewriting Strategy to Generate Prime Implicates in Equational Logic. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'14)*. Springer, 2014
6. M. Echenim, N. Peltier, and S. Tourret. An approach to abductive reasoning in equational logic. In *Proceedings of IJCAI'13 (International Conference on Artificial Intelligence)*, pages 3–9. AAAI, 2013
7. M. Echenim, N. Peltier, and S. Tourret. A Superposition Strategy for Abductive Reasoning in Ground Equational Logic. In *Proceedings of IWS 2012 (International Workshop on Strategies)*, 2012
8. M. Echenim and N. Peltier. Reasoning on Schemata of Formulae. In *Proceedings of CICM 2012 (Conferences on Intelligent Computer Mathematics)*, volume 7362, pages 310–325. Springer LNCS, 2012
9. M. Echenim and N. Peltier. A Calculus for Generating Ground Explanations. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'12)*, volume 7364, pages 194–209. Springer LNCS, 2012
10. M. Echenim and N. Peltier. Instantiation of SMT problems modulo Integers. In *AISC 2010 (10th International Conference on Artificial Intelligence and Symbolic Computation)*, volume 6167 of *LNCS*, pages 49–63. Springer, 2010
11. Thierry Boy de la tour, Mnacho Echenim, and Paliath Narendran. Unification and matching modulo leaf-permutative equational presentations. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *4th International Joint Conference, IJCAR 2008*, LNAI 5195, pages 332–347, Sydney, Australia, August 2008. Springer Verlag
12. Maria Paola Bonacina and Mnacho Echenim. T-decision by decomposition. In Frank Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 199–214. Springer, 2007
13. Maria Paola Bonacina and Mnacho Echenim. Rewrite-based satisfiability procedures for recursive data structures. *Electr. Notes Theor. Comput. Sci.*, 174(8):55–70, 2007

14. Maria Paola Bonacina and Mnacho Echenim. Rewrite-based decision procedures. *Electr. Notes Theor. Comput. Sci.*, 174(11):27–45, 2007

Travaux en cours/non publiés

1. Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. Prime Implicate Generation in Equational Logic. 2016
2. Mnacho Echenim and Nicolas Peltier. Combining induction and saturation-based theorem proving. 2016
3. Mnacho Echenim and Nicolas Peltier. Generating prime implicates by decomposition. 2016
4. Mnacho Echenim and Nicolas Peltier. On the construction of equational binary decision diagrams for equational formulæ. 2015
5. Thierry Boy de la Tour, Mnacho Echenim, and Nicolas Peltier. Boolean abductive reasoning with symmetries. 2014
6. Thierry Boy de la Tour and Mnacho Echenim. Solving linear constraints in elementary abelian p-groups of symmetries. *CoRR*, abs/1107.4553, 2011

Documents pédagogiques

1. Pierre Berlioux, Mnacho Echenim et Michel Lévy : Théorie des langages. Polycopié destiné aux étudiants de première année (niveau L3) à Grenoble INP-Ensimag.
2. Thierry Boy de la Tour et Mnacho Echenim : Éléments de logique pour le cours de deuxième année Ensimag : Fondements de Logique pour l'Informatique. Polycopié destiné aux étudiants de deuxième année (niveau M1) à Grenoble INP-Ensimag.

Travaux encadrés

2016- : Co-encadrement (avec N. Peltier) du doctorat de Yanis Sellami : *Abductive Reasoning Modulo Theories.*

2015-2016 : Co-encadrement (avec N. Peltier) du stage de M2R de Yanis Sellami : *A DPLL-based approach to prime implicate generation.*

2015 : Co-encadrement (avec N. Peltier) du stage d'Introduction à la Recherche en Laboratoire (Ensimag 2A) de Yanis Sellami : *Dénombrement de modèles en logique équationnelle.*

2014 : Co-encadrement (avec N. Peltier) du stage d'Assistant Ingénieur de Rémi Galan-Alfonso et Raphaël Bayle : *Détection de symétries dans les problèmes SMT.*

2012-2016 : Co-encadrement (avec N. Peltier) du doctorat de Sophie Tourret : *Prime Implicate Generation in Equational Logic.*

2011-2012 : Co-encadrement (avec N. Peltier) du stage de M2R de Sophie Tourret : *Abduction and prime implicates, from propositional logic to equational logic.*

2011 : Co-encadrement (avec N. Peltier) du stage d'Assistant Ingénieur de Sophie Tourret : *Abduction modulo des théories.*

2011 : Co-encadrement (avec N. Peltier) du stage de Travaux d'Etude et de Recherche de Sophie Tourret : *Extraction de noyaux insatisfaisables dans les problèmes SMT.*

2010 : Co-encadrement (avec N. Peltier) du stage de Travaux d'Etude et de Recherche de Ludovic Queiroga : *Preuve par instanciation en déduction automatique.*

Responsabilités administratives

2016- : Chargé de mission de l'axe *Méthodes formelles* au Laboratoire d'Informatique de Grenoble.

2011- : Responsable de l'équipe pédagogique *Bases théoriques de l'Informatique* à Grenoble INP-Ensimag.

2010-2016 : Membre élu du Conseil d'École de l'Ensimag.

2009- : Co-responsable de la filière *Ingénierie pour la Finance* à Grenoble INP-Ensimag.

Projets

2016-2018 : Co-porteur d'un projet de maturation de la SATT de Saclay.

2010-2013 : Participation au projet ANR ASAP : *About Schemata and Proof.*

2008-2009 : Participation au projet ANR ARROWS : *Safe Pointer-Based Data Structures : A Declarative Approach to their Specification and Analysis.*

Chapitre 3

Démonstrateurs automatiques et problèmes SMT

3.1 Présentation

Une des façons de s'assurer qu'un système – comme par exemple un logiciel – vérifie bien certains critères ou spécifications est de construire une formule logique qui est valide si et seulement si le système vérifie les critères en question. Cette approche permet donc de garantir qu'un système ne contient aucune erreur en testant la validité (ou, de façon duale, la satisfaisabilité) de la formule construite. Dans certains cas, les traductions peuvent s'opérer vers des formules de logique propositionnelle, pour lesquelles il existe des outils de résolution très efficaces, mais en général, la complexité des systèmes considéré nécessite l'emploi de logiques plus expressives pour obtenir une traduction plus compacte et souvent plus naturelle. Nous nous intéressons ici plus particulièrement au cas où le système peut être traduit en une formule en logique équationnelle du premier ordre. Les formules considérées sont généralement des ensembles de clauses qui se décomposent en deux parties : un ensemble de clauses non fermées définissant les propriétés des objets apparaissant dans les formules, comme par exemple les propriétés des structures de données utilisées dans un programme, tels les listes ou les tableaux ; et un ensemble de clauses fermées traduisant le comportement du système. Le premier ensemble de clauses peut être considéré comme un ensemble d'axiomes définissant une théorie, et le problème consiste alors à trouver un modèle de l'ensemble de clauses traduisant le comportement du système qui ne contredit pas la théorie. Il s'agit donc de tester la satisfaisabilité d'un ensemble de clauses fermées *modulo* une théorie, d'où l'appellation générale de *problème de satisfaisabilité modulo des théories*, ou *problème SMT*. Un outil capable de résoudre de tels problèmes pour une théorie $\mathcal{T}$ donnée est appelé une *procédure de $\mathcal{T}$-décision* (voir par exemple [20]).

Dans le cas général, cette approche est évidement limitée puisque la lo-

gique du premier ordre est semi-décidable : si une formule est valide, ceci peut être démontré automatiquement, mais si elle ne l'est pas, il n'y a aucune garantie qu'un démonstrateur puisse le détecter. De nombreux travaux de recherche portent sur l'identification de classes de formules ou de restrictions de structures pour lesquelles il existe des procédures de décision, c'est-à-dire des programmes capables de décider si une formule de cette classe est valide ou non. Plusieurs résultats classiques ont permis d'identifier certaines de ces classes et restrictions de structures, comme le calcul des prédicats du premier ordre monadique [119], la classe de Bernays-Schönfinkel-Ramsey [23, 150], la classe d'Ackermann [2] ou encore l'arithmétique de Presburger [147, 162]. D'autres classes ont été identifiées par la communauté de chercheurs travaillant sur les problèmes SMT, qui ont prouvé que ces problèmes sont décidables pour de nombreuses théories, dont celles définissant les structures de données employées dans des programmes, telles les listes, les tableaux ou les tableaux de bits. En complément à la recherche de procédures de décision pour des théories données, de nombreux travaux ont été consacrés à la mise en place d'implémentations efficaces de ces procédures, capables de supporter une montée en charge. En effet, les formules logiques décrivant le comportement de systèmes sont souvent de taille très importante, et nécessitent des millions de symboles. En général, la taille importante des formules considérées est due à leur structure combinatoire, et une première approche a consisté en l'utilisation des outils qui ont été mis au point pour tester la satisfaisabilité de formules en logique propositionnelle pour résoudre les problèmes SMT. Le principe de cette approche est basé sur la conversion du problème SMT à tester en une formule propositionnelle équisatisfaisable, avant de faire appel à un solveur SAT pour en tester la satisfaisabilité. Les travaux menés sur cette thématique portaient principalement sur la mise au point des algorithmes de conversion les plus efficaces possibles afin que les tailles des problèmes SMT et de leurs traductions propositionnelles restent du même ordre de grandeur [19]. L'intérêt de cette approche est que, une fois la transformation opérée, il est possible de se servir de n'importe quel solveur SAT pour résoudre le problème considéré. Ceci permet donc d'exploiter des années de recherche sur la mise au point de solveurs SAT efficaces ; les solveurs SAT les plus performants étant capables de résoudre efficacement des problèmes comprenant des milliers de variables et des millions de clauses. Cette approche a néanmoins deux inconvénients majeurs : dans de nombreux cas, la conversion du problème SMT en problème SAT peut résulter en une formule propositionnelle quadratiquement plus large que celle d'entrée, pour laquelle même les solveurs SAT les plus efficaces ne peuvent pas fournir de réponse en un temps raisonnable. Un autre inconvénient est que la traduction en formule propositionnelle peut rompre la structure du problème d'origine et en rendre sa résolution plus compliquée.

Afin de remédier à ces problèmes, il est possible d'adopter une technique consistant à intercaler des phases de raisonnement sur la partie proposition-

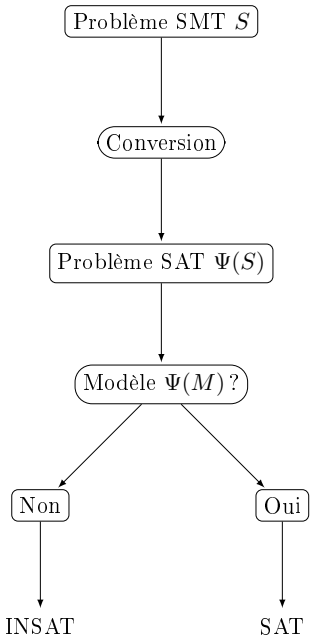

FIGURE 3.1 – Conversion SMT → SAT.

nelle des problèmes SMT avec des phases de raisonnement sur la théorie sous-jacente : c'est l'approche DPLL($\mathcal{T}$) [139]. Conceptuellement, la phase de raisonnement sur la partie propositionnelle du problème est basée sur l'algorithme DPLL [61, 60], et plus précisément sur son raffinement CDCL (*conflict-driven clause learning*) [180] ; tandis que la phase de raisonnement sur la théorie sous-jacente est effectuée par un outil capable de décider si une conjonction de littéraux est satisfaisable modulo cette théorie. Cette approche met donc en interaction deux outils : un solveur SAT et une procédure dite de $\mathcal{T}$-*satisfaisabilité*. Etant donné un problème SMT S dont la théorie sous-jacente est $\mathcal{T}$, l'algorithme procède de la façon suivante (voir aussi la Figure 3.2) :

- La formule S est abstraite en une formule propositionnelle $\Psi(S)$, en remplaçant chaque atome par un symbole propositionnel. Si la formule $\Psi(S)$ est insatisfaisable, alors il est clair que S l'est également.
- Un solveur SAT cherche à trouver un modèle de $\Psi(S)$. S'il n'en existe aucun, l'algorithme renvoie INSAT.
- Si le solveur exhibe un modèle $M' \stackrel{\text{def}}{=} \Psi(M)$, M' étant donc un ensemble de littéraux propositionnels, la procédure de $\mathcal{T}$-satisfaisabilité est invoquée sur l'ensemble M constitué des littéraux dont M' est l'abstraction. Si M est $\mathcal{T}$-satisfaisable, alors l'algorithme renvoit SAT car un modèle a été trouvé pour le problème de départ.
- Si la procédure de $\mathcal{T}$-satisfaisabilité renvoit INSAT, alors l'abstraction

$\Psi(S)$ est raffinée afin d'empêcher M' d'en être un modèle, et le solveur SAT est à nouveau invoqué.

Pour construire un solveur SMT pour une théorie $\mathcal{T}$ donnée avec cette approche, il suffit donc d'implémenter une procédure de $\mathcal{T}$-satisfaisabilité à faire interagir avec un solveur SAT. Evidemment, la conception d'une procédure de $\mathcal{T}$-satisfaisabilité est loin d'être triviale, puisqu'il faut démontrer que la procédure est correcte (si elle renvoit INSAT alors la formule en entrée est insatisfaisable modulo $\mathcal{T}$) et complète (si elle renvoit SAT, alors il existe un modèle de la formule modulo $\mathcal{T}$). Il faut de plus s'assurer qu'aucune erreur d'implémentation n'a été introduite lors du développement de la procédure. Une grande majorité des solveurs SMT modernes sont néanmoins basés sur cette approche et sur ses améliorations, comme par exemple l'extraction par la procédure de $\mathcal{T}$-satisfaisabilité du plus d'information possible quand un ensemble de littéraux n'est pas satisfaisable, afin de raffiner l'abstraction propositionnelle considérée le plus efficacement possible. D'autres techniques ont été mises au point pour gagner en efficacité. Ainsi, les procédures de $\mathcal{T}$-satisfaisabilité modernes sont incrémentales : si une telle procédure a été invoquée avec une conjonction donnée de littéraux et que la procédure doit à nouveau être invoquée avec la même conjonction à laquelle un littéral a été ajouté, alors le nouveau test peut être effectué à moindre coût. Une telle caractéristique permet une interaction plus forte entre le solveur SAT et la procédure de $\mathcal{T}$-satisfaisabilité, puisque la recherche d'un modèle peut être effectuée incrémentalement, à chaque fois que le solveur SAT exhibe un littéral qui fait partie du modèle potentiel. Les procédures de $\mathcal{T}$-satisfaisabilité modernes sont également capables d'effectuer des retours sur trace, afin de pouvoir par exemple tester des conjonctions de littéraux qui ne diffèrent que d'un élément sans devoir refaire l'intégralité des calculs à chaque fois. L'approche DPLL($\mathcal{T}$) est adoptée par la grande majorité des solveurs SMT récents, dont les performances peuvent être comparées lors de compétitions [18], sur des benchmarks accessibles à tous [1] [17].

Se posent alors deux questions :

- Est-il possible d'utiliser des procédures de $\mathcal{T}$-satisfaisabilité existantes quand la théorie considérée est une combinaison, comme par exemple l'union de deux théories plus simples ? Ainsi, peut-on tester qu'une formule est satisfaisable modulo la combinaison de la théorie des tableaux et celle des listes en se servant des procédures de $\mathcal{T}$-satisfaisabilité des théories individuelles ?
- Est-il nécessaire d'implémenter une nouvelle procédure de $\mathcal{T}$-satisfaisabilité quand la théorie considérée est une extension d'une théorie plus simple ? Par exemple, si une formule encode (à l'aide de quantificateurs) le fait que tous les éléments d'un tableau sont distincts deux à deux, est-il nécessaire d'implémenter une procédure de $\mathcal{T}$-satisfaisabilité

1. http://smtlib.cs.uiowa.edu/

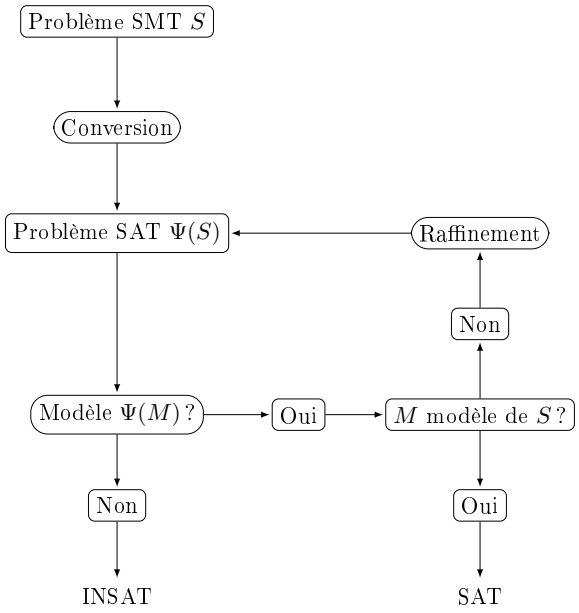

FIGURE 3.2 – Approche DPLL($\mathcal{T}$).

spécifique pour les tableaux dont tous les éléments sont distincts deux à deux ?

Ces questions surviennent naturellement puisque par exemple, il est fréquent d'avoir des problèmes de vérification sur des programmes manipulant simultanément plusieurs structures de données différentes, et de spécifier que certaines propriétés doivent être vérifiées sur ces structures (un exemple est la spécification qu'un tableau doit être trié, ce qui nécessite de faire interagir la théorie décrivant les tableaux et celle décrivant la relation d'ordre sur ses éléments). Une réponse standard à la première question est l'emploi des algorithmes de combinaison proposés par Nelson et Oppen [136] ou Shostak [159]. Ces algorithmes permettent de faire interagir les procédures de $\mathcal{T}$-satisfaisabilité de plusieurs théories pour tester la satisfaisabilité d'une formule modulo l'union de ces théories. Ils ont été conçus dans le cadre où les théories considérées ne contiennent aucun symbole en commun autre que le prédicat de l'égalité et sont de stable-infinité [2]. De nombreux travaux ont étendu ces algorithmes à des cas plus généraux, dans lesquels les théories peuvent avoir des symboles en commun ou encore ne pas être de stable-infinité [167, 96, 179, 91, 13, 53]. D'autres travaux de recherche ont pour but de répondre par la négative à la deuxième question en ajoutant aux solveurs SMT des procédures d'instanciation. Le principe est de

2. Une théorie $\mathcal{T}$ est de stable-infinité si toute formule $\mathcal{T}$-satisfaisable admet un modèle de domaine infini.

remplacer les clauses contenant des variables autres que les axiomes de base de la théorie considérée par un nombre suffisant d'instances pour garantir que l'ensemble de clauses obtenu est satisfaisable exactement quand le problème d'origine est satisfaisable. Comme il est possible qu'il n'existe aucune procédure d'instanciation complète pour une théorie donnée, de nombreux solveurs SMT emploient des heuristiques afin d'instancier les clauses avec des variables [66, 64, 94, 70], mais certains travaux de recherche portent sur l'identification de classes de formules pour lesquelles des procédures d'instanciation complètes peuvent être employées en complément de procédures de $\mathcal{T}$-satisfaisabilité [44, 95].

Utilisation de démonstrateurs automatiques

Une façon conceptuellement simple de répondre aux deux questions du paragraphe précédent est de se servir d'un démonstrateur automatique pour la logique équationnelle du premier ordre. Pour tester si une formule est satisfaisable modulo une théorie $\mathcal{T}$ qui est définissable par un ensemble fini d'axiomes, il suffit de fournir au démonstrateur cette formule avec les axiomes définissant $\mathcal{T}$. Si $\mathcal{T}$ est l'union de plusieurs théories, alors il suffit de fournir au démonstrateur l'union des axiomes définissant chacune des théories, et si la formule en entrée contient des clauses avec des variables, ces clauses sont ajoutées au démonstrateur de façon transparente. Comme un démonstrateur automatique est complet pour la réfutation, il est garanti que si le problème SMT fourni en entrée est insatisfaisable, alors le démonstrateur en construira une réfutation. Si la formule SMT est satisfaisable, alors le démonstrateur automatique engendrera un ensemble saturé dont peut être extrait un modèle de la formule. Il se peut cependant que l'ensemble saturé soit infini, dans quel cas le démonstrateur ne terminera pas. Pour garantir qu'un démonstrateur peut être utilisé comme un solveur SMT pour une théorie donnée, il faut donc s'assurer que, quelque soit l'ensemble de clauses fermées fourni en entrée, l'ensemble saturé construit par le démonstrateur est fini.

Les démonstrateurs automatiques pour la logique équationnelle du premier ordre les plus efficaces sont basés sur le calcul de Superposition $\mathcal{SP}$ [140]. Ce calcul peut être vu comme une extension du calcul de résolution [118] qui intègre le raisonnement équationnel de façon efficace, notamment grâce à la réécriture et à des restrictions sur les inférences possibles, basées par exemple sur des relations d'ordre. Des travaux ont eu pour but d'identifier des théories pour lesquelles les démonstrateurs basés sur le calcul de Superposition peuvent être utilisés comme procédures de $\mathcal{T}$-satisfaisabilité, et plusieurs théories ont été identifées, par exemple dans [11, 10]. Les théories identifiées permettent principalement de définir des structures de données dans des programmes, mais d'autres théories comme celle définissant les homomorphismes ou bien celle définissant les ensembles finis peuvent être traitées par des démonstrateurs automatiques. Dans [10], les auteurs

ont également identifié une condition suffisante portant sur les théories, la variable-inactivité, qui garantit qu'un démonstrateur qui termine sur des théories variable-inactives, termine également sur leur combinaison. Intuitivement, ceci peut s'expliquer par le fait que toute théorie variable-inactive est de stable-infinité [36, 121]. D'autres travaux ont eu pour but de généraliser ces résultats, que ce soit en définissant des transformations de problèmes pour se servir d'un démonstrateur dans le cas de théories définies par des ensembles infinis d'axiomes [10], ou en définissant des procédures permettant de tester automatiquement des résultats de terminaison d'une part [120], et de combinaison d'autre part [121, 171]. Des études théoriques et expérimentales ont également été menées pour évaluer l'efficacité des procédures de $\mathcal{T}$-satisfaisabilité résultantes [11, 10]. D'un point de vue théorique, une mesure de cette efficacité est l'estimation du rapport entre le nombre de clauses engendrées dans le pire cas par le démonstrateur et le nombre de clauses dans l'ensemble de départ. Ce rapport peut être exponentiel, c'est le cas par exemple pour la théorie des tableaux [3], mais il est polynomial pour de nombreuses théories, ce qui garantit qu'un démonstrateur automatique peut être employé comme procédure de $\mathcal{T}$-satisfaisabilité efficace pour ces dernières.

3.2 Décidabilité et complexité

3.2.1 $\mathcal{T}$-satisfaisabilité de structures de données récursives et acycliques

Il est possible de se servir d'un démonstrateur automatique pour tester la satisfaisabilité d'une formule modulo une théorie $\mathcal{T}$ définie par un ensemble infini d'axiomes en associant au problème d'origine un ensemble fini de clauses sur lequel le démonstrateur termine, et qui est satisfaisable si et seulement si le problème d'origine est $\mathcal{T}$-satisfaisable. Nous avons étudié dans [32] l'application de cette technique au cas des structures de données récursives et acycliques. Pour un paramètre k donné, ces structures sont définies par l'ensemble infini $\mathcal{R}$ d'axiomes suivant :

$$\begin{aligned} sel_i(cons(x_1, \ldots, x_k)) &\simeq x_i \quad \text{pour } i = 1, \ldots, k, \\ cons(sel_1(x), \ldots, sel_k(x)) &\simeq x, \\ t[x] &\not\simeq x, \end{aligned}$$

où t représente un terme construit sur les symboles $sel_1, \ldots, sel_k$, qui contient une occurrence de x ; notons Ac l'ensemble des axiomes de la forme $t[x] \not\simeq x$. Nous avons construit un algorithme permettant de transformer tout problème de $\mathcal{T}$-satisfaisabilité dans cette théorie en un ensemble *fini* de clauses qui lui est équisatisfaisable. Cette transformation se déroule en deux étapes :

3. Ce résultat est prévisible car le problème de satisfaisabilité modulo la théorie des tableaux est NP-complet [67]

– Nous avons introduit l'axiome d'extensionalité suivant :

$$\mathbf{ext} : \bigwedge_{i=1}^{k} (sel_i(x) \simeq sel_i(y)) \Rightarrow x \simeq y,$$

et montré comment transformer un ensemble de clauses fermées S en un ensemble de clauses fermées S' tel que $\mathcal{R} \cup S$ est satisfaisable si et seulement si $\{\mathbf{ext}\} \cup \mathrm{Ac} \cup S'$ l'est également. La transformation que nous avons définie est linéaire en la taille de S.

– Nous avons prouvé qu'il existe un entier n au plus égal à la taille de S, tel que si $\mathrm{Ac}[n]$ représente l'ensemble des éléments de Ac de la forme $t[x] \not\simeq x$ où t est de longueur au plus n, alors $\{\mathbf{ext}\} \cup \mathrm{Ac} \cup S'$ et $\{\mathbf{ext}\} \cup \mathrm{Ac}[n] \cup S'$ sont équisatisfaisables.

Enfin, nous avons démontré que le calcul de Superposition termine sur tout ensemble de la forme $\{\mathbf{ext}\} \cup \mathrm{Ac}[n] \cup S'$, prouvant ainsi qu'un démonstrateur automatique peut être employé comme procédure de $\mathcal{T}$-satisfaisabilité pour la théorie des structures de données récursives et acycliques.

3.2.2 Des procédures de $\mathcal{T}$-satisfaisabilité polynomiales

Bien qu'un démonstrateur automatique puisse être utilisé comme procédure de $\mathcal{T}$-satisfaisabilité pour de nombreuses théories, l'efficacité de la procédure résultante n'était pas satisfaisante pour certaines d'entre elles. C'est le cas par exemple pour la théorie des *integer offsets*, qui est une structure de données récursive acyclique avec un constructeur d'arité 1 et un seul sélecteur. Un démonstrateur utilisé pour résoudre des problèmes dans cette théorie engendre un nombre exponentiel de clauses, alors qu'il existe des procédures de $\mathcal{T}$-satisfaisabilité polynomiales pour cette théorie (voir par exemple [138]). Une étude détaillée des inférences réalisées par un démonstrateur sur les problèmes dans cette théorie nous a permis d'affiner la transformation présentée dans le paragraphe précédent, et de fournir au démonstrateur un ensemble de clauses dont la taille est linéaire en celle du problème d'origine, et pour lequel il est garanti que le démonstrateur engendre un nombre polynomial de clauses. Nous avons ainsi prouvé dans [34] qu'un démonstrateur automatique pouvait être utilisé comme procédure de $\mathcal{T}$-satisfaisabilité polynomiale pour cette théorie.

Nous avons également défini dans [34] une procédure de $\mathcal{T}$-satisfaisabilité polynomiale pour la théorie des enregistrements avec extension. La théorie des enregistrements, paramétrée par k est définie par l'ensemble $\mathcal{E}$ d'axiomes suivant :

$$\begin{array}{lcll} rselect_i(rstore_i(x, v)) & \simeq & v & \text{pour } 1 \leq i \leq k \\ rselect_i(rstore_j(x, v)) & \simeq & rselect_i(x) & \text{pour } 1 \leq i \neq j \leq k \end{array}$$

La théorie des enregistrements avec extension est définie par les axiomes

ci-dessus et par l'axiome d'extensionnalité

$$\bigwedge_{i=1}^{k}(rselect_i(y) \simeq rselect_i(y)) \Rightarrow x \simeq y.$$

Alors que les problèmes de $\mathcal{T}$-satisfaisabilité peuvent être résolus en temps polynomial pour la théorie des enregistrements [10], il n'en est pas de même pour la théorie des enregistrements avec extension, où un nombre exponentiel de clauses peuvent être engendrées. Nous avons défini une tranformation du problème d'origine qui permet de résoudre les mêmes problèmes en temps polynomial. La transformation décompose le problème d'origine S en un ensemble S_1 de clauses unitaires positives et un ensemble S_2 de clauses négatives tels que S est satisfaisable modulo la théorie des enregistrements avec extensionnalité si et seulement si $\mathcal{E} \cup S_1 \cup S_2$ est satisfaisable. Nous avons montré que le calcul de Superposition appliqué à $\mathcal{E} \cup S_1$ engendre un nombre polynomial de clauses qui sont toutes unitaires, et qu'il est possible d'en extraire en temps polynomial un sous-ensemble F de clauses fermées tel que $\mathcal{E} \cup S_1 \cup S_2$ et $F \cup S_2$ sont équisatisfaisables. Ce dernier ensemble ne contient que des clauses de Horn, et sa satisfaisabilité peut donc être testée en temps polynomial.

3.3 Démonstrateurs et procédures de $\mathcal{T}$-décision

Nous nous sommes intéressés à la question suivante : pour quelles théories est-il possible de se servir d'un démonstrateur automatique pour construire une procédure de $\mathcal{T}$-décision ? Quand le démonstrateur est une procédure de $\mathcal{T}$-satisfaisabilité, une solution simple à ce problème est de considérer une forme normale disjonctive $\bigvee_{i=1}^{n} F_i$ équivalente à S, et d'invoquer le démonstrateur sur chaque ensemble [4] $\mathcal{T} \cup F_i$. Il est cependant clair que cette solution n'est pas efficace. Nous avons donc d'abord étudié pour quelles théories il est garanti que le démonstrateur terminera sur des entrées comprenant les axiomes de la théorie et un ensemble quelconque de clauses fermées, qui ne sont pas nécessairement unitaires.

L'approche employée dans [11, 10] pour prouver que $\mathcal{SP}$ est une procédure de $\mathcal{T}$-satisfaisabilité consiste à énumérer les différentes formes des clauses qui peuvent être engendrées par le système d'inférence. S'il y a un nombre fini de catégories de clauses et que chacune de ces catégories ne peut contenir qu'un ensemble fini de clauses, ceci signifie que tout démonstrateur implémentant $\mathcal{SP}$ termine pour tout problème de $\mathcal{T}$-satisfaisabilité. Cette méthode est difficile à étendre aux problèmes de $\mathcal{T}$-décision, pour lesquels il faut démontrer la terminaison d'un démonstrateur non plus pour un ensemble de clauses unitaires fermées, mais pour un ensemble de clauses fermées

4. Nous confondons ici une conjonction de littéraux avec l'ensemble de clauses unitaires correspondant.

quelconque. De plus, il doit être répété pour chaque théorie, et ces énumérations à la main peuvent être longues et sources d'erreurs. C'est pourquoi dans les travaux décrits ci-dessous, nous avons cherché des caractérisations syntaxiques des classes de formules sur lesquelles les techniques proposées peuvent être utilisées.

Aplatissement d'ensembles de clauses. Afin de démontrer des résultats de terminaison sur des systèmes d'inférence, il est nécessaire de pouvoir contrôler les inférences qui sont effectuées. C'est pour ceci que dans ce qui suit, les ensembles de clauses fournis en entrée sont systématiquement *aplatis*. La procédure d'aplatissement consiste en l'introduction de nouveaux symboles de constantes qui ont pour rôle de désigner des termes de profondeur non-nulle qui apparaissent dans l'ensemble d'origine, et le remplacement de ces sous-termes par la constante les désignant. Par exemple, si S est l'ensemble $\{f(f(a)) \simeq b \vee g(f(a)) \not\simeq f(b)\}$, alors l'ensemble suivant est un aplatissement de S :

$$\{f(a) \simeq c_1,\ f(c_1) \simeq c_2,\ g(c_1) \simeq c_3,\ f(b) \simeq c_4,\ c_2 \simeq b \vee c_3 \not\simeq c_4\}\,.$$

Cette opération permet de décomposer tout ensemble de clauses fermées S en :

- Un ensemble de clauses unitaires S_f, chaque clause dans S_f étant de la forme $f(a) \simeq b$;
- Un ensemble de clauses *plates* S_0, chaque terme apparaissant dans S_0 étant une constante.

L'ensemble S_f correspond à un ensemble de *définitions* des constantes nouvellement introduites, et l'ensemble S_0 contient la structure booléenne du problème d'origine. Pour tout ensemble fermé S et toute théorie $\mathcal{T}$, l'ensemble $\mathcal{T} \cup S$ est satisfaisable si et seulement si $\mathcal{T} \cup S_f \cup S_0$ est satisfaisable.

3.3.1 Utilisation d'un démonstrateur uniquement

Une première façon de s'assurer qu'un démonstrateur automatique peut être utilisé comme procédure de $\mathcal{T}$-décision a été étudiée dans [31]. Cette approche est basée sur le constat que, pour la plupart des théories employées dans la communauté SMT, les axiomes définissant la théorie peuvent être séparés en deux catégories :

- Les axiomes définissant les propriétés des symboles de fonction interprétés ;
- Les axiomes spécifiant la façon dont interagissent ces symboles de fonction interprétés.

Prenons par exemple la théorie des tableaux avec extensionalité. Cette théorie comporte deux symboles de fonctions interprétés : store, un symbole d'arité 3 qui permet d'insérer un élément dans un tableau, et select, un

$$\forall x, z, v.\ \text{select}(\text{store}(x, z, v), z) \simeq v \quad (3.1)$$
$$\forall x, z, v, w.\ z \simeq w \vee \text{select}(\text{store}(x, z, v), w) \simeq \text{select}(x, w) \quad (3.2)$$
$$\forall x, y.\ (\forall z.\text{select}(x, z) \simeq \text{select}(y, z)) \Rightarrow x \simeq y \quad (3.3)$$

FIGURE 3.3 – Axiomes de la théorie des tableaux avec extensionalité

symbole d'arité 2 qui permet de lire la valeur d'un élément stocké dans un tableau. Dans cette théorie, deux tableaux dont les éléments stockés à des indices identiques sont identiques, doivent nécessairement être égaux. Formellement, cette théorie est définie par les axiomes de la Figure 3.3. L'axiome 3.3 spécifie une propriété de la fonction select, et les axiomes 3.1 et 3.2 définissent les interactions entre select et store.

Une instance d'un problème de $\mathcal{T}$-décision peut alors être représentée par un triplet $\langle T_f, T_d, T_i \rangle$, où T_f est un ensemble de clauses fermées et $\mathcal{T} = T_d \cup T_i$. Chaque clause dans T_d représente une propriété d'un symbole de fonction, et chaque clause dans T_i représente une propriété d'interaction entre deux symboles de fonctions. Nous avons montré que pour certaines théories, il est possible d'associer à chaque problème de $\mathcal{T}$-décision un tel triplet et que toutes les clauses engendrées par le calcul de Superposition sont dans un de ces ensembles. C'est le cas par exemple de la théorie des tableaux. Si $S = \{\text{store}(a, i, e) \simeq b\}$, alors $\mathcal{SP}$ appliqué à l'axiome (3.1) et à la clause dans S engendre la clause fermée $\text{select}(b, i) \simeq e$ qui est dans T_f, et $\mathcal{SP}$ appliquée à l'axiome (3.2) et à la clause dans S engendre la clause $i \simeq w \vee \text{select}(b, w) \simeq \text{select}(a, w)$ qui spécifie une propriété du symbole select, et est donc dans T_d.

Nous avons étudié précisément pour quelles théories l'existence des trois ensembles finis de clauses est garantie. Nous avons défini un ensemble de conditions sur chaque élément de ce triplet ; tout triplet vérifiant l'ensemble des conditions est *sous-terme inactif*, et une théorie $\mathcal{T}$ est *sous-terme inactive* si elle peut être décomposée en un triplet sous-terme inactif. Intuitivement, les conditions imposées aux différents éléments du triplet permettent de contrôler les inférences du calcul de Superposition, et permettent de garantir des propriétés de finitude qui impliquent la terminaison du calcul sur l'ensemble de clauses de départ. La propriété d'inactivité des sous-termes n'est pas impactée par l'ajout d'un ensemble de clauses fermées : plus précisément, si $\langle T_f, T_d, T_i \rangle$ est sous-terme inactif et $S_f \cup S_0$ est un aplatissement de S, alors $\langle S_f \cup S_0 \cup T_f, T_d, T_i \rangle$ est également sous-terme inactif. La propriété d'inactivité des sous-termes permet donc de garantir qu'un démonstrateur implémentant le calcul de Superposition peut être utilisé comme procédure de $\mathcal{T}$-décision. Plusieurs théories, dont la théorie des tableaux et certaines de ses extensions [31], les structures de données récursives et les ensembles finis

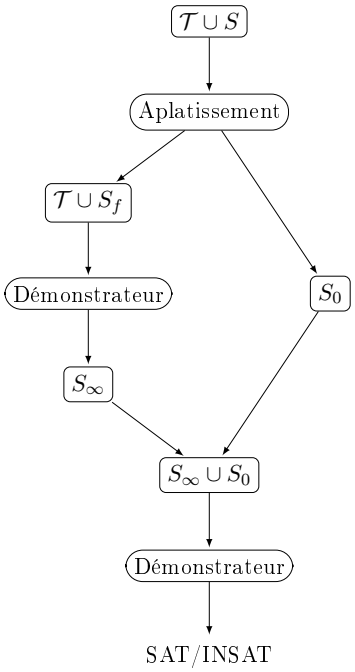

FIGURE 3.4 – Une procédure de $\mathcal{T}$-décision basée sur un démonstrateur.

[11] sont sous-terme inactives. Une caractéristique importante des conditions faisant qu'une théorie est sous-terme inactive est que, même si elles sont nombreuses, une seule d'entre elles n'est pas syntaxique et ne peut pas être vérifiée automatiquement ; et cette condition doit également être vérifiée dans l'approche employée dans [11, 10].

La notion de sous-terme inactivité apporte une première réponse à la façon dont il est possible de garantir qu'un démonstrateur automatique peut être utilisé dans une procédure de $\mathcal{T}$-décision. Cette solution n'est cependant pas entièrement satisfaisante. Une raison principale est que, comme expliqué précédemment, la conversion d'un ensemble de clauses en formule en forme normale disjonctive permet de se servir d'une procédure de $\mathcal{T}$-satisfaisabilité dans une procédure de $\mathcal{T}$-décision. Ceci signifie qu'intuitivement, on peut s'attendre à ce qu'un démonstrateur qui peut être utilisé comme procédure de $\mathcal{T}$-satisfaisabilité puisse être utilisé comme procédure de $\mathcal{T}$-décision. Or, un démonstrateur peut être utilisé comme procédure de $\mathcal{T}$-satisfaisabilité pour certaines théories, comme la théorie des listes éventuellement vides, qui ne sont pas sous-terme inactives. De plus, bien que la plupart des conditions que doit vérifier une théorie sous-terme inactive peuvent l'être automatiquement, ces conditions interagissent de façon complexe et il est difficile de savoir si certaines d'entre elles peuvent être relâchées pour obtenir une classe de théories plus large. Nous avions démontré que toute théorie sous-terme

inactive est également variable-inactive. La propriété de variable-inactivité avait été identifiée dans [10], comme condition suffisante de modularité : un démonstrateur qui est à la fois une procédure de $\mathcal{T}_1$-satisfaisabilité et de $\mathcal{T}_2$-satisfaisabilité est une procédure de $(\mathcal{T}_1 \cup \mathcal{T}_2)$-satisfaisabilité à chaque fois que $\mathcal{T}_1$ et $\mathcal{T}_2$ sont variable-inactives. Nous avons montré dans [34] comment transformer une procédure de $\mathcal{T}$-satisfaisabilité en procédure de $\mathcal{T}$-décision quand $\mathcal{T}$ est variable-inactive. Le principe est le suivant (voir la Figure 3.4). Après aplatissement de la formule d'entrée S, le démonstrateur est invoqué sur l'ensemble $\mathcal{T} \cup S_f$. Par hypothèse, le démonstrateur est une procédure de $\mathcal{T}$-satisfaisabilité et termine, engendrant l'ensemble de clauses persistantes S_∞. Le démonstrateur est à nouveau invoqué sur l'ensemble $S_\infty \cup S_0$. Cet ensemble est satisfaisable si et seulement si $\mathcal{T} \cup S$ l'est également, et nous avons prouvé que le démonstrateur termine nécessairement sur $S_\infty \cup S_0$ quand $\mathcal{T}$ est variable-inactive. Ce résultat permet donc de prouver qu'un démonstrateur automatique qui peut être utilisé comme procédure de $\mathcal{T}$-satisfaisabilité peut également être utilisé comme procédure de $\mathcal{T}$-décision quand $\mathcal{T}$ est variable-inactive. C'est le cas de toutes les théories considérées dans [11, 10, 31, 32] pour lesquelles ce résultat est donc valable.

3.3.2 Démonstrateur faiblement couplé à un solveur SMT

Une autre façon dont un démonstrateur automatique qui est une procédure de $\mathcal{T}$-satisfaisabilité peut être utilisé dans une procédure de $\mathcal{T}$-décision est d'employer la stratégie de DPLL($\mathcal{T}$) en couplant ce démonstrateur à un solveur SAT. Les travaux sur cette méthode ont été motivés par deux limitations des démonstrateurs pour la résolution de problèmes SMT. La première est que certaines théories particulièrement importantes pour les problèmes SMT, et en particulier l'arithmétique, ne pouvaient pas être gérées par un démonstrateur, même si plusieurs travaux ont permis l'intégration de fragments de l'arithmétique à des démonstrateurs génériques [4, 116, 22, 59]. La seconde est liée à l'efficacité des procédures de $\mathcal{T}$-décision obtenues en utilisant un démonstrateur automatique seul. En effet, les démonstrateurs ne sont pas conçus pour gérer des structures combinatoire complexes de façon efficace, ce qui a été confirmé par des études expérimentales. Cependant, faire interagir ainsi un démonstrateur et un solveur SAT est loin d'être évident car, pour obtenir un outil efficace, il est nécessaire de fortement coupler ces composants. Cette approche a néanmoins été employée dans [65, 151, 173] et les différents auteurs ont mis au point des outils dans lesquels un démonstrateur automatique est entièrement intégré à un solveur SAT.

Nous avons exploré dans [33, 35] une nouvelle approche permettant d'obtenir une procédure de $\mathcal{T}$-décision en couplant *faiblement* un démonstrateur automatique à un solveur SMT. Le principe de ce couplage faible repose sur la procédure d'aplatissement. Intuitivement, quand un ensemble de clauses est aplati, seule la partie définitionnelle de l'ensemble résultant interagit

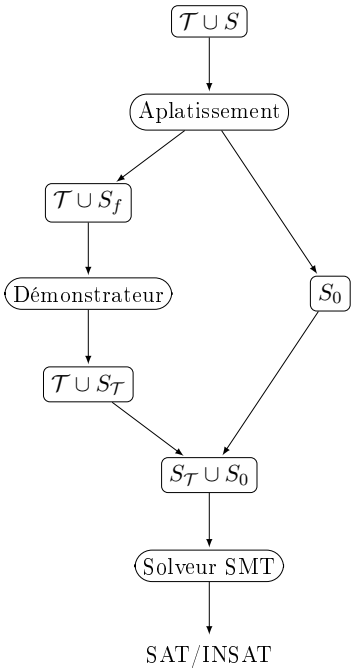

FIGURE 3.5 – Compilation d'une théorie.

avec la théorie considérée. Etant donné un problème à résoudre de la forme $\mathcal{T} \cup S_0 \cup S_f$, nous avons exploré sous quelles conditions l'ensemble saturé qui est engendré par un démonstrateur prenant en entrée l'ensemble $\mathcal{T} \cup S_0$ –et qui est de la forme $\mathcal{T} \cup S_\mathcal{T}$– est tel que $\mathcal{T} \cup S_0 \cup S_f$ et $S_\mathcal{T} \cup S_f$ sont équisatisfaisables. Cette approche permettrait donc de se débarasser des axiomes de $\mathcal{T}$, et donc d'obtenir un problème SMT potentiellement plus simple à résoudre. Nous avons ainsi défini la notion de $\mathcal{T}$-stabilité et prouvé que cette approche s'applique à de nombreuses théories, dont celles des tableaux, des enregistrements et des integer offsets. Cette approche permet aussi de combiner des théories de deux façons différentes : soit en fournissant au démonstrateur automatique l'union des axiomes définissant les théories, soit en compilant les théories *séparément* (voir la Figure 3.6).

3.4 Discussion

La dernière approche proposée offre de nombreux avantages. Elle permet une séparation claire des tâches, la gestion des clauses contenant des variables étant déléguée au démonstrateur automatique, tandis que la structure combinatoire des problèmes est gérée par des solveurs SMT. Le couplage entre le démonstrateur étant étant faible, il est en pratique possible d'utiliser n'im-

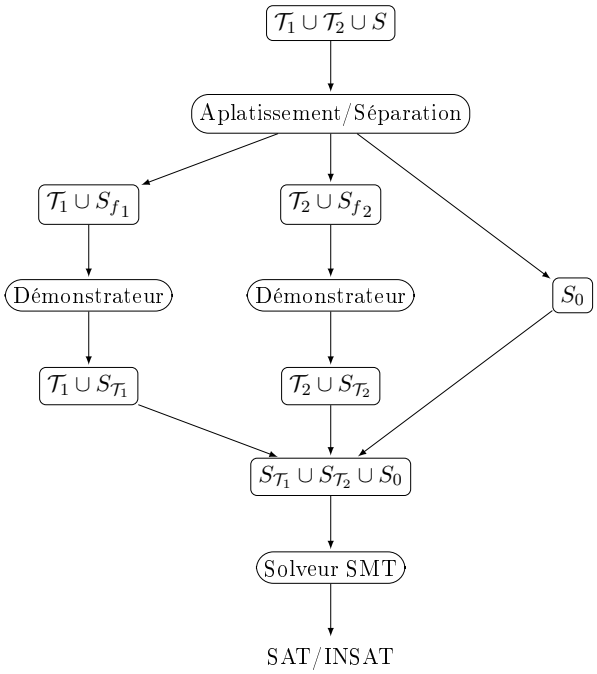

FIGURE 3.6 – Compilation de plusieurs théories.

porte quel démonstrateur et de le faire interagir avec n'importe quel solveur SMT pour un coût de développement réduit. La compilation de la théorie par le démonstrateur automatique permet de s'en débarasser entièrement, et cette approche peut donc être vue comme une procédure de réduction. Ainsi, pour des théories comme celle des enregistrements ou celle des integer offsets, les ensembles $S_{\mathcal{T}}$ obtenus sont fermés et le solveur SMT doit simplement tester la satisfaisabilité d'un ensemble de clauses fermées en logique du premier ordre avec égalité. Pour la théorie des tableaux, les ensembles compilés doivent être résolus modulo la théorie plus simple des fonctions partielles, dont les axiomes spécifient que certaines fonctions sont égales, sauf sur un ensemble défini d'élements. Il est donc envisageable, avec cette approche, de focaliser le développement de solveurs SMT pour un nombre réduit de théories, et de laisser le démonstrateur automatique réduire les problèmes de $\mathcal{T}$-satisfaisabilité dans des théories plus complexes vers des problèmes dans ces théories plus simples.

Chapitre 4

Des procédures d'instanciation

4.1 Présentation

Un intérêt majeur de la méthode décrite dans [35] est qu'elle permet un traitement uniforme de problèmes SMT dans plusieurs théories à l'aide d'un démonstrateur automatique faiblement couplé à un solveur SMT capable de traiter certaines théories de base. Différents aspects de cette méthode sont néanmoins améliorables. Ainsi, comme il n'y a pas de caractérisation syntaxique des théories pour lesquelles cette technique peut être employée, il n'est pas toujours évident de vérifier son applicabilité. De plus, cette approche peut parfois nécessiter un post-traitement. C'est le cas pour les problèmes dans la théorie des tableaux, qui sont transformés par cette méthode en problèmes dans la théorie plus simple des fonctions partielles. Afin de pouvoir se servir d'un solveur SMT qui ne serait pas spécifiquement conçu pour cette théorie, une procédure d'instanciation a été mise au point dans [35] afin d'obtenir un problème équisatisfaisable, ne contenant que des clauses fermées et pouvant donc être résolu par n'importe quel solveur SMT capable de gérer la logique équationnelle – une fonctionnalité de base pour un solveur SMT. Enfin, des études expérimentales ont montré que, pour certaines théories dont celle des tableaux, un démonstrateur automatique n'est pas capable de tenir la charge quand la conjonction de littéraux fournie en entrée contient des milliers d'éléments.

De nombreuses procédures de $\mathcal{T}$-satisfaisabilité sont basées sur une transformation du problème d'origine S afin d'engendrer un problème équisatisfaisable S' constitué uniquement de clauses fermées, sans mention à une théorie quelconque [44, 164]. De façon plus ou moins implicite, ces transformations sont basées sur l'instanciation des axiomes de la théorie considérée et sur des propriétés qui sont spécifiques à chaque théorie. Les concepteurs de ces procédures doivent évidemment démontrer que $\mathcal{T} \cup S$ et S' sont équisatisfaisables en raisonnant sur le lien entre les clauses engendrées et la théorie sous-jacente. L'approche de [35] a également pour but d'engendrer un en-

semble équisatisfaisable S', mais cette fois sans qu'il soit nécessaire de raisonner sur une théorie spécifique, l'emploi d'un démonstrateur automatique garantissant l'équisatisfaisabilité de $\mathcal{T} \cup S$ et S'. L'ensemble de clauses S' est engendré par les règles d'inférences du calcul de Superposition, qui peuvent être interprétées comme des transformations basées sur la réécriture et la Résolution, appliquées à des *instances* des axiomes de $\mathcal{T}$. Nous avons cherché à identifier les instances des axiomes de $\mathcal{T}$ qui sont employées dans la génération de S', afin de mettre au point une procédure d'instanciation Θ telle que $\mathcal{T} \cup S$ et $\Theta(S)$ sont équisatisfaisables.

Dans le cas général, une procédure d'instanciation est une fonction qui, à un ensemble de clauses S, associe un ensemble de clauses fermées $\Theta(S)$. Dans la plupart des cas, les clauses de $\Theta(S)$ sont des instances de celles dans S, ce qui rend ces procédures trivialement correctes : si $\Theta(S)$ est insatisfaisable, alors S l'est également. Une procédure d'instanciation est complète pour la réfutation si, pour tout ensemble de clauses S, les ensembles S et $\Theta(S)$ sont équisatisfaisables. Le théorème de Herbrand garantit l'existence de procédures d'instanciations complètes pour la réfutation en logique du premier ordre, puisque tout ensemble de clauses insatisfaisable admet un ensemble d'instances fini qui est également insatisfaisable. La conception d'une procédure d'instanciation complète nécessite donc de s'assurer que les clauses de l'ensemble de départ sont suffisamment instanciées pour détecter l'insatisfaisabilité, tout en veillant à ne pas considérer un nombre trop important d'instances pour que la procédure soit efficace. Il existe également des procédures d'instanciation *modulo* des théories : ces procédures engendent un ensemble de clauses fermées $\Theta(S)$ tel que S et $\Theta(S)$ sont équisatisfaisables modulo la théorie en question. S'il existe plusieurs procédures d'instanciations complètes pour la réfutation [117, 144, 93, 21], ces procédures ne terminent pas nécessairement car dans le cas où S est satisfaisable, il est possible que la procédure engendre un ensemble infini de clauses. Il existe néanmoins des procédures d'instanciation qui sont complètes et terminent pour des classes de formules particulières. Il en existe par exemple pour les formules qui admettent un ensemble fini d'instances fermées, comme la classe de Bernays-Schönfinkel-Ramsey, constituée des formules universelles sans aucun symbole de fonction autre que des constantes ; ou bien pour la classe des ensembles de clauses stratifiées [1]. D'autres classes ont été identifées dans un cadre plus proche des problèmes SMT : les ensembles de ces classes sont de la forme $\mathcal{T} \cup S$, où S est un ensemble de clauses fermées. Des procédures ont été développées dans le cas où $\mathcal{T}$ est la théorie des tableaux [43, 97], la théorie de structures avec pointeurs telles que les listes [128], la théorie des ensembles avec cardinalité [141], ou encore une théorie locale [98, 161, 109].

Nous avons étudié l'emploi de procédures d'instanciation pour dans un cadre suffisamment général pour inclure les problèmes SMT dans de nombreuses théories. Le but est similaire à celui recherché avec l'emploi de démonstrateurs automatiques : les procédures mises au point doivent permettre

de résoudre des problèmes SMT, quantifiés ou non, dans de nombreuses théories, en se servant uniquement de solveurs SMT capables de gérer quelques théories de base. L'objet des travaux n'est pas nécessairement d'obtenir des procédures d'instanciation complètes pour la réfutation, mais de garantir que les outils résultants sont efficaces et capables de supporter une montée en charge en instanciant les clauses non fermées le moins possible.

4.2 Une procédure d'instanciation générique

Nous avons défini dans [82] une procédure d'instanciation qui, partant d'un ensemble de clauses quelconque S, engendre un ensemble $\Theta_{\mathrm{g}}(S)$ de clauses fermées qui est fini. Cette procédure a été mise au point après avoir analysé les instanciations effectuées par le calcul de superposition lors de la résolution de problèmes SMT à l'aide d'un démonstrateur automatique. Cette analyse avait permis de constater que pour chacune des théories pour lesquelles un démonstrateur automatique peut être employé comme procédure de $\mathcal{T}$-satisfaisabilité, un pré-traitement du problème d'origine et un choix judicieux d'ordre sur les termes permet de garantir que dans toute dérivation, les axiomes de la théorie ne sont instanciés que par les termes apparaissant dans l'ensemble de clauses de départ. Autrement dit, pour de nombreuses théories fréquemment utilisées par la communauté SMT, une procédure qui en instancie les axiomes avec les termes fermés apparaissant dans le problème d'origine engendre un ensemble de clauses fermées fini tout en étant complète pour la réfutation. L'ensemble de clauses engendré pouvant être très large et dans un souci d'efficacité, nous avons imposé des conditions, essentiellement basées sur l'unification, pour restreindre les instances à considérer. Intuitivement, les instances retenues sont celles obtenues par la *pseudo-unification* de certains termes ; ceci correspond à une forme d'unification de termes avec le même symbole de tête, dont les arguments peuvent *potentiellement* être rendus égaux au cours d'une dérivation. Par exemple, si les littéraux $f(a, x) \simeq a$ et $f(c, b) \simeq c$ apparaissent dans le problème d'origine S, alors bien que $f(a, x)$ et $f(c, b)$ ne soient pas unifiables, ils peuvent le devenir si c est remplacé par a après application de la règle de superposition. Ainsi, l'instance $f(a, b) \simeq a$ du premier littéral est retenue dans la construction de l'ensemble $\Theta_{\mathrm{g}}(S)$. Plus généralement, pour un ensemble de clauses donné S, la règle d'instanciation proposée engendre des instances de la forme $C\sigma$, où C est une clause de S et σ est un pseudo-unificateur d'un terme apparaissant dans C et d'un terme apparaissant dans une autre clause de S. Les pseudo-unificateurs associent toujours une constante à une variable, ce qui garantit que la procédure termine.

Exemple 1 Considérons l'ensemble de clauses constitué des éléments sui-

vants :

$$\begin{array}{llll} 1: & \texttt{cons}(\texttt{car}(x), \texttt{cdr}(x)) \simeq x, & 2: & \texttt{car}(\texttt{cons}(x, y)) \simeq x, \\ 3: & \texttt{cdr}(\texttt{cons}(x, y)) \simeq y, & 4: & \texttt{car}(a) \simeq b, \\ 5: & \texttt{cons}(a, c) \simeq d, & 6: & \texttt{car}(\texttt{cdr}(b)) \simeq c. \end{array}$$

Les trois premières clauses sont une axiomatisation de la théorie des listes non-vides. La procédure d'instanciation engendre les clauses supplémentaires décrites dans le tableau ci-dessous.

Termes	Clauses	Instanciations
$\texttt{car}(x), \texttt{car}(a)$	(1), (4)	$\texttt{cons}(\texttt{car}(a), \texttt{cdr}(a)) \simeq a$
$\texttt{cdr}(x), \texttt{cdr}(b)$	(1), (6)	$\texttt{cons}(\texttt{car}(b), \texttt{cdr}(b)) \simeq b$
$\texttt{cons}(x, y), \texttt{cons}(a, c)$	(2), (5)	$\texttt{car}(\texttt{cons}(a, c)) \simeq a$
$\texttt{cons}(x, y), \texttt{cons}(a, c)$	(3), (5)	$\texttt{cdr}(\texttt{cons}(a, c)) \simeq c$

La première colonne du tableau contient les termes sur lesquels est invoqué l'algorithme de pseudo-unification, la seconde colonne contient les numéros des clauses dans lesquelles apparaissent ces termes, et la troisième colonne contient les clauses obtenues. ♣

La procédure est étendue en permettant l'ajout aux instances de *conditions d'accord*, qui représentent intuitivement des égalités entre constantes qui, si elles sont vérifiées, rendent unifiables les termes employés dans l'algorithme de pseudo-unification. Par exemple, partant de l'ensemble de clauses

$$\begin{array}{llll} 1: & p(a, x) \simeq \top \vee q(x, x) \not\simeq \top & 2: & p(c, d) \not\simeq \top \\ 3: & q(f(x), x) \simeq \top & 4: & a \simeq b \\ 5: & b \simeq c & 6: & f(x) \simeq x \end{array}$$

la procédure engendre les clauses suivantes (nous précisons à chaque fois les clauses concernées et le terme fermé employé pour la pseudo-unification) :

$$\begin{array}{lll} 7 & p(a, d) \simeq \top \vee q(d, d) \not\simeq \top \vee a \not\simeq c & 1, 2,\ p(c, d) \\ 8 & q(f(d), d) \simeq \top & 3, 7,\ q(d, d) \\ 9 & f(d) \simeq d & 6, 8,\ f(d) \end{array}$$

Dans la clause 7, la condition d'accord $a \not\simeq c$ est ajoutée à l'instance de la clause 1 : elle traduit le fait que si les constantes a et c sont égales, alors $p(a, x)$ et $p(c, d)$ sont unifiables. L'ajout de ces conditions d'accord augmente potentiellement la taille de l'ensemble de clauses fermées engendré par la procédure, mais un solveur SMT pourrait potentiellement résoudre le problème correspondant plus efficacement grâce à ces conditions, car elles peuvent permettre de constater qu'une clause n'est pas nécessaire pour construire une réfutation. Ainsi, un solveur qui dans son état courant considère un modèle partiel pour lequel a et c sont différents, éliminera la clause 7 lors de la recherche d'une réfutation.

Pour une théorie $\mathcal{T}$ fixée, la procédure proposée engendre un nombre *polynomial* d'instances de ses axiomes. Ceci est un avantage considérable sur la procédure de compilation de [35] qui se sert d'un démonstrateur automatique ; par exemple, pour la théorie des tableaux, cette procédure d'instanciation engendre un nombre quadratique d'instances tandis que la procédure de compilation engendre un nombre exponentiel de clauses fermées. Cette procédure ne peut cependant pas être complète pour la réfutation puisqu'elle engendre toujours un ensemble fini d'instances.

Nous avons néanmoins démontré que la procédure est complète pour des ensembles de clauses *simplement prouvables*. Ce critère sémantique est lié aux dérivations qui peuvent être effectuées sur l'ensemble de clauses en question avec le calcul de superposition, et ne dépend donc d'aucune théorie en particulier. Nous avons ensuite étudié quels critères sur les ensembles de clauses fournis en entrée garantissent que ces ensembles sont simplement prouvables. Nous avons ainsi défini la classe des ensembles $\mathcal{C}$*-contrôlables* pour lesquels la propriété de simple prouvabilité est garantie. Les critères que doivent vérifier les ensembles sont tous syntaxiques, et nous avons réalisé une implémentation permettant de tester automatiquement si un ensemble est $\mathcal{C}$-contrôlable ou non [1]. Un avantage de cette approche pour la résolution de problèmes SMT est qu'il n'est pas nécessaire de systématiquement tester la $\mathcal{C}$-contrôlabilité sur chaque ensemble de la forme $\mathcal{T} \cup S$ afin de vérifier que l'ensemble de clauses engendrées par la procédure d'instanciation et l'ensemble $\mathcal{T} \cup S$ seront équisatisfaisables. En effet, la classe des ensembles $\mathcal{C}$-contrôlables admet la propriété de stabilité suivante : si S est $\mathcal{C}$-contrôlable et S' est un ensemble de clauses aplaties, alors $S \cup S'$ est également $\mathcal{C}$-contrôlable. C'est pourquoi il suffit de tester *une fois pour toutes* si une théorie est $\mathcal{C}$-contrôlable ou non, pour garantir, le cas échéant, que la procédure d'instanciation sera complète. Nous avons ainsi pu vérifier automatiquement que *toutes* les théories étudiées dans [11, 10], ainsi que leurs unions, sont $\mathcal{C}$-contrôlables.

Nous avons mené une évaluation expérimentale de l'efficacité de cette procédure en comparant les performances d'un solveur SMT sur des problèmes SMT, se servant ou non d'une phase d'instanciation. Nous avons sélectionné une des séries de tests disponible sur SMT-LIB [2] concernant la théorie des tableaux. Les problèmes de cette série permettent de vérifier que si deux tableaux sont identiques à l'origine, et que des éléments y sont insérés aux mêmes indices distincts mais dans des ordres différents, alors les tableaux résultants sont à nouveau identiques. Nous avons également créé de nouvelles séries de problèmes, pour exprimer le fait qu'une matrice symétrique le reste si les éléments de sa diagonale sont remplacés par des valeurs arbitraires, ou si des éléments symétriques sont remplacés simultanément par la même valeur, et pour exprimer le fait que les projections de tableaux multidimensionnels

1. http://lig-membres.imag.fr/peltier/fish.html
2. http://smtlib.cs.uiowa.edu/

ont une intersection non-vide. Nous avons résolu les problèmes de ces séries en nous servant du solveur SMT Yices[3]. Ce solveur SMT contient une procédure de décision spécifique pour la théorie des tableaux, nous avons donc pu comparer les temps d'exécution de l'outil quand les problèmes SMT sont fournis tels quels, ou quand ce sont des instances fermées qui sont fournies en entrée. Il y a a priori deux façons de se servir de la procédure d'instanciation.

- Celle qui serait potentiellement la plus efficace consiste à la coupler fortement au solveur SMT, de telle sorte à instancier les variables de façon paresseuse. Cette façon permettrait d'éviter qu'un nombre trop important de termes fermés soient considérés dans la phase d'instanciation.
- L'autre façon consiste à instancier les variables une fois pour toute, avant même d'invoquer un solveur SMT. Cette façon de procéder est nettement plus simple, car elle ne nécessite aucun couplage particulier avec le solveur SMT qui peut être utilisé comme une boîte noire.

Nous avons choisi cette seconde façon de procéder. Le détail des résultats sont disponibles dans [82]. Ils montrent que la procédure d'instanciation permet au solveur SMT de gagner en efficacité pour la théorie des tableaux, mais qu'elle est moins efficace pour les problèmes que le solveur est capable de traiter en employant ses propres heuristiques d'instanciation. Ce résultat était prévisible puisque les heuristiques d'instanciation des solveurs sont principalement focalisées sur l'efficacité et n'offrent aucune garantie de complétude. L'intérêt de notre procédure devient claire sur les problèmes pour lesquels les instanciations à effectuer ne sont pas immédiates à détecter. Sans cette procédure d'instanciation, le solveur Yices n'est capable de résoudre aucune instance du problème d'intersection des tableaux multidimensionnels, mis à part la plus triviale, tandis qu'en faisant appel à la procédure d'instanciation, le solveur est capable de résoudre des problèmes jusqu'en dimension 5 en moins de 300 secondes.

4.3 Un traitement des entiers naturels

Si la procédure d'instanciation Θ_g présentée dans [82] est générique et peut donc être employée pour de nombreuses théories, elle est clairement inapplicable sur celles qui sont définies par un ensemble infini d'axiomes. C'est le cas par exemple pour l'arithmétique de Presburger, alors que cette théorie apparaît fréquemment dans les problèmes SMT : le dépôt de problèmes SMT-LIB contient ainsi de nombreuses instances tirées de cas concrets portant sur des tableaux dont les indices sont des entiers et sur des propriétés qui sont exprimées dans l'arithmétique de Presburger. Nous avons étudié dans [71] de quelle façon construire une procédure d'instanciation modulo l'arithmétique de Presburger pour des problèmes contenant certaines variables de sorte en-

3. http://yices.csl.sri.com/

tière, et d'autres non. Ce processus se déroule en deux étapes, et consiste à d'abord instancier les variables entières afin d'obtenir un problème équisatisfaisable ne contenant aucune variable de sorte entière, avant d'instancier les variables restantes à l'aide d'une autre procédure d'instanciation. Cette technique et la démonstration de complétude de la procédure résultante sont basées sur la *méthode de raisonnement différé* de [16], qui a également été employée pour intégrer de façon efficace le raisonnement arithmétique dans des démonstrateurs automatiques standard [4].

La séparation des étapes de raisonnement sur la partie arithmétique d'une formule et sur le reste de la formule se fait grâce à des *clauses contraintes*. Une clause contrainte est de la forme [4] $[\![C \mid \Lambda]\!]$, où C est une clause standard dont les seuls termes de sorte entière sont des variables, et Λ est un ensemble de contraintes arithmétiques, c'est-à-dire d'atomes de la forme $t \simeq s$ ou bien $t \preccurlyeq s$, où s et t sont des termes de sorte entière (et représentent donc des entiers). A n'importe quelle clause standard peut être associée une clause contrainte.

Exemple 2 Soit $C = g(a) \simeq 0 \vee f(a) \not\simeq b \vee h(b) \simeq c$ et supposons que les termes a et $g(a)$ sont de sorte entière. Alors la clause contrainte $[\![f(x) \not\simeq b \vee h(b) \simeq c \mid g(a) \not\simeq 0, a \simeq x]\!]$ est équivalente à C. ♣

Nous avons adapté le calcul de Superposition pour tenir compte des clauses contraintes, et défini une version de ce calcul dans laquelle les contraintes des prémisses sont propagées aux conclusions de chaque règle d'inférence. Cette méthode permet de séparer le raisonnement sur les termes entiers du reste du raisonnement, comme l'illustre l'exemple suivant.

Exemple 3 Considérons l'ensemble S de clauses suivant :

$$\{p(0),\ \neg p(x) \vee p(\mathtt{s}(x)),\ \neg p(n)\}\,,$$

où $\mathtt{s}$ est la fonction successeur, et n est de sorte entière. L'abstraction correspondant à cet ensemble de clauses est

$$\{[\![p(x) \mid x \simeq 0]\!]\,,\ [\![\neg p(x) \vee p(\mathtt{s}(x)) \mid \top]\!]\,,\ [\![\neg p(y) \mid y \simeq n]\!]\}\,.$$

La règle de superposition appliquée à la première et troisième clause contrainte engendre $[\![\Box \mid n \simeq 0]\!]$, ce qui signifie que n ne peut pas être égal à 0. La règle de superposition appliquée aux deux premières clauses de S engendre la clause contrainte $[\![p(y) \mid y \simeq \mathtt{s}(0)]\!]$ qui, avec la troisième clause de S engendre $[\![\Box \mid n \simeq \mathtt{s}(0)]\!]$, ce qui signifie que n ne peut pas être égal à 1. Il est aisé de vérifier que le calcul permet d'engendrer l'ensemble des clauses de la forme $\left[\!\!\left[\Box \;\middle|\; n \simeq \mathtt{s}^k(0)\right]\!\!\right]$ pour tout $k \in \mathbb{N}$, et cet ensemble est insatisfaisable modulo l'arithmétique de Presburger. ♣

4. La notation employée ici n'est pas la même que celle de [71]. Le changement a été fait pour des raisons d'uniformisation des notations.

Comme le montre l'exemple ci-dessus, l'adaptation du calcul de Superposition ne permet pas d'obtenir une procédure de décision, puisqu'il peut être nécessaire d'engendrer un ensemble infini de clauses avant de conclure [5]. Nous avons défini une classe d'ensemble de clauses pour lesquelles le calcul est complet en restreignant la forme des atomes pouvant apparaître dans les contraintes, définissant ainsi les *clauses précontraintes*. Il est garanti que si une règle d'inférence du calcul de Superposition adapté est appliqué à deux clauses précontraintes, alors la clause engendrée est également précontrainte. Nous avons prouvé que le calcul appliqué à un ensemble de clauses précontraintes insatisfaisable modulo l'arithmétique de Presburger engendre un ensemble *fini* de clauses contraintes de la forme $\{[\![\Box \mid \Lambda_i]\!] \mid i = 1, \ldots, n\}$ tel que $\bigwedge_{i=1}^{n} \Lambda_i$ est insatisfaisable modulo l'arithmétique de Presburger. Nous avons également montré comment identifier à partir de l'ensemble de clauses fourni en entrée un ensemble B de termes fermés de sorte entière tels que, si S' est obtenu en instanciant les variables de sorte entière par les éléments de B, alors S et S' sont équisatisfaisables. Afin d'obtenir une procédure d'instanciation efficace, nous avons cherché à construire un ensemble B contenant le moins d'éléments possible. Nous avons ainsi pu montrer que, pour les tableaux avec des indices entiers, la procédure d'instanciation résultante engendre au plus autant d'instances que celle de [43], et que dans certains cas, elle engendre exponentiellement moins d'instances tout en demeurant complète pour la réfutation. Des expérimentations avec le démonstrateur Z3 [6] ont permis de confirmer que ce dernier résout les problèmes sur les tableaux avec des indices entiers plus rapidement quand ils sont instanciés avec notre procédure.

La procédure mise au point permet donc d'instancier les variables de sorte entière apparaissant dans un ensemble de clauses S pour obtenir un ensemble S' sans variable de sorte entière, tel que S et S' sont équisatisfaisables modulo l'arithmétique de Presburger. Cet ensemble peut néanmoins encore contenir des variables d'autres sortes, et nous avons étudié sous quelles conditions il est garanti qu'une autre procédure d'instanciation permet à partir de S' d'engendrer un ensemble S'' de clauses fermées tel que S et S'' sont équisatisfaisables modulo l'arithmétique de Presburger. Nous avons démontré qu'une condition suffisante pour assurer cette propriété est que la procédure d'instanciation Θ soit :

- monotone : si $S \subseteq S'$ alors $\Theta(S) \subseteq \Theta(S')$;
- stable par atomes équationnels : si s et t sont des termes fermés tels que tout terme non variable apparaissant dans S qui est unifiable avec s (resp. t) est égal à s (resp. t), alors $\Theta(S \cup \{s \simeq t\}) = \Theta(S) \cup \{s \simeq t\}$.

C'est le cas de la procédure générique définie dans [82] ; les approches de [82]

5. Par exemple, l'ensemble $\{[\![\Box \mid n \simeq \mathtt{s}^k(0)]\!] \mid k \in \mathbb{N}\}$ est insatisfaisable, mais chacun de ses sous-ensembles finis est satisfaisable.

6. https://z3.codeplex.com/

et [71] permettent donc, en utilisant un solveur SMT pour la logique équationnelle et l'arithmétique de Presburger, de traiter n'importe quel problème SMT qui associe l'arithmétique de Presburger et une théorie pour laquelle la méthode de [82] est complète, à condition que les clauses fournies en entrée soient des clauses précontraintes.

4.4 Combinaisons de procédures d'instanciation

4.4.1 Instanciation d'une union d'ensembles

Nous nous sommes intéressés à la combinaison de procédures d'instanciation, afin de résoudre des problèmes dans des unions de théories dans le cas où il existe des procédures d'instanciation pour les théories individuelles. De par sa généricité, la procédure définie admet une propriété importante pour résoudre des problèmes dans des unions de théories : quand deux ensembles de clauses sont $\mathcal{C}$-contrôlables, Θ_g est une procédure d'instanciation complète pour leur union. Dans un contexte plus général, nous avons étudié dans [81] de quelle façon combiner des procédures d'instanciation Θ_1 et Θ_2 qui sont respectivement complètes (éventuellement modulo des théories) pour deux classes $\mathcal{C}_1$ et $\mathcal{C}_2$ d'ensembles de clauses, afin d'obtenir une procédure complète pour des ensembles de clauses de la forme $S_1 \cup S_2$, où $S_1 \in \mathcal{C}_1$ et $S_2 \in \mathcal{C}_2$. Nous nous sommes placés dans le cas où $\mathcal{C}_1$ et $\mathcal{C}_2$ ne partagent aucun symbole de fonction autre que des constantes, ce cadre est donc très proche de celui adopté pour la combinaison de procédures de décision basées sur l'approche de Nelson-Oppen.

La façon la plus naturelle de construire une telle procédure est de considérer la fonction $\Theta_1 \sqcup \Theta_2$, telle que

$$\Theta_1 \sqcup \Theta_2(S_1 \cup S_2) \stackrel{\text{def}}{=} \Theta_1(S_1) \cup \Theta_2(S_2).$$

Cependant, la fonction ainsi définie n'est pas complète en général, et nous avons identifié un ensemble de conditions afin de garantir la complétude de cette procédure.

- La première condition concerne les cardinalités des domaines associés à chaque sorte : deux classes $\mathcal{C}_1$ et $\mathcal{C}_2$ sont *structurellement équipotentes* quand pour toute sorte $\mathtt{s}$ commune à $\mathcal{C}_1$ et $\mathcal{C}_2$, et pour tous ensembles satisfaisables $S_1 \in \mathcal{C}_1$ et $S_2 \in \mathcal{C}_2$, il existe des modèles I_1 et I_2 de ces ensembles tels que $I_1(\mathtt{s})$ et $I_2(\mathtt{s})$ sont de même cardinalité. Cette condition peut être vue comme une extension de la notion de stable-infinité [166] et est similaire à l'approche employée dans [168] pour combiner les procédures de programmation logique par contraintes.
- La seconde condition concerne les procédures d'instanciation elles-mêmes et est proche de celle que nous avons identifiée dans [71] et décrite dans le paragraphe précédent.

Nous avons défini la notion de *complétude à la base* d'une procédure d'instanciation : une procédure Θ est complète à la base si pour tout ensemble de clauses S et tout ensemble E d'équations et diséquations entre constantes, les ensembles $S \cup E$ et $\Theta(S) \cup E$ sont équisatisfaisables. Nous avons prouvé que si $\mathcal{C}_1$ et $\mathcal{C}_2$ sont structurellement équipotentes et Θ_1 et Θ_2 sont complètes à la base, alors $\Theta_1 \sqcup \Theta_2$ est une procédure d'instanciation complète pour tout ensemble de la forme $S_1 \cup S_2$, où $S_1 \in \mathcal{C}_1$ et $S_2 \in \mathcal{C}_2$.

Ce résultat apporte donc une réponse à la façon dont des procédures d'instanciation peuvent être combinées pour obtenir une nouvelle procédure qui est complète pour la réfutation. Si les procédures de [82, 71, 95] sont complètes à la base, ce n'est évidemment pas le cas de toutes les procédures d'instanciation. Nous avons alors montré comment, partant d'une procédure d'instanciation Θ complète et monotone – pour tout ensemble de clauses S et pour toutes constantes a, b, $\Theta(S) \subseteq \Theta(S \cup \{a \simeq b\})$ –, il est possible de construire une procédure d'instanciation $\Theta^\diamond$ qui est complète à la base, à condition que les instances engendrées par Θ soient indépendantes des diséquations entre constantes, une propriété qui se traduit par : $\Theta(S \cup \{a \not\simeq b\}) = \Theta(S) \cup \{a \not\simeq b\}$.

4.4.2 Combinaison hiérarchique

La méthode générique décrite précédemment permet de faire interagir des procédures d'instanciation pour des classes individuelles pour construire une procédure d'instanciation pour l'union d'éléments de ces classes. Cette méthode ne peut pas être utilisée pour construire une procédure d'instanciation pour des ensembles dont les clauses peuvent contenir des symboles interprétés dans chacune des classes, alors que de tels ensembles apparaissent fréquemment dans les problèmes SMT. Par exemple, une clause exprimant qu'un tableau t est constant sur l'intervalle $[m..n]$ est :

$$m \not\leq x \vee x \not\leq n \vee \text{select}(t, x) \simeq c.$$

Cette clause non fermée contient à la fois des symboles interprétés dans l'arithmétique de Presburger (le symbole d'inégalité $\leq$), et dans la théorie des tableaux (le symbole select de lecture dans un tableau).

Nous avons considéré dans [83] le cas où l'ensemble des sortes sur lesquelles le problème est construit peut être séparé en deux ensembles disjoints :

- L'ensemble des *sortes de base*, qui sont telles que chacun des arguments d'une fonction dont le codomaine est une sorte de base est également d'une sorte de base ;
- L'ensemble des *sortes englobantes*, qui contient les sortes qui ne sont pas de base.

Les ensembles de clauses que nous avons considérés sont appelés des *ensembles hiérarchiques*, ils contiennent des clauses de la forme[7] $C^B \vee C^N$, où tous les termes dans C^B sont d'une sorte de base, et tous les termes dans C^N sont d'une sorte englobante, avec la restriction supplémentaire que les seuls sous-termes d'une sorte de base dans C^N sont des variables. Cette condition sur les sous-termes d'une sorte de base n'est pas contraignante, car il est aisé de remplacer une clause ne vérifiant pas cette condition par une clause équivalente qui la vérifie. Par exemple, considérons deux sortes $\mathtt{s}$ et $\mathtt{s}'$, où $\mathtt{s}$ est une sorte de base et $\mathtt{s}'$ est une sorte englobante, ainsi que les symboles $a : \mathtt{s}$, $b : \mathtt{s}'$ et $f : \mathtt{s} \times \mathtt{s}' \to \mathtt{s}'$. Alors la clause unitaire $f(a, b) \simeq b$ peut être remplacée par la clause équivalente $x \not\simeq a \vee f(x, b) \simeq b$ qui vérifie la propriété en question. Intuitivement, ce cadre impose une hiérarchie entre les termes des différentes sortes : les termes de base permettent d'exprimer des contraintes sur les termes d'une sorte englobante. Par exemple, la clause exprimant que le tableau t est constant entre les bornes m et n est construite sur deux sortes : la sorte entière qui est la sorte de base pour les indices du tableau, et les sortes pour les tableaux eux-mêmes et leurs éléments, qui sont des sortes englobantes. La clause peut être décomposée en une clause de base $m \not\leq x \vee x \not\leq n$ exprimant une contrainte sur les indices du tableau, et une clause $\mathrm{select}(t, x) \simeq c$ exprimant que le tableau est constant.

Etant donné un ensemble de clauses hiérarchiques S qui est de la forme $\{C_i^B \vee C_i^N \mid i = 1, \ldots, n\}$, nous avons cherché à déterminer comment, étant données deux procédures d'instanciation Θ_B et Θ_N, respectivement complètes pour $\{C_i^B \mid i = 1, \ldots, n\}$ et $\{C_i^N \mid i = 1, \ldots, n\}$, construire une procédure d'instanciation $\Theta_N[\Theta_B]$ complète pour S. Le calcul hiérarchique de [16] permet de séparer les étapes de raisonnements dans les différentes théories, et de la même façon, la hiérarchie imposée dans notre cadre permet de séparer les phases d'instanciation pour les différentes classes d'ensembles de clauses. L'algorithme que nous avons mis au point est le suivant :

1. Dans l'ensemble $\{C_i^N \mid i = 1, \ldots, n\}$, chaque variable d'une sorte de base est instanciée par un symbole de constante arbitraire $\bullet$ de même sorte. On note S_N l'ensemble ainsi obtenu, qui ne peut contenir que des variables de sortes englobantes.
2. La procédure Θ_N est invoquée sur S_N et engendre un ensemble d'instances fermées des clauses dans S_N.
3. Les substitutions qui ont engendré des instances fermées de clauses de S_N à l'étape précédente sont utilisées pour instancier les clauses dans l'ensemble S de départ. On obtient ainsi un ensemble S_B d'instances de S, ne contenant que des variables d'une sorte de base.
4. La procédure Θ_B appliquée à l'ensemble $\{C_i^B \mid i = 1, \ldots, n\}$ instancie chaque variable d'une sorte de base par un ensemble de termes fer-

7. La lettre N en exposant de la partie englobante s'explique par le fait que dans [83], nous avons employé le terme "Nested", traduit ici par "Englobante".

més. Toutes les variables de base ainsi que les symboles $\bullet$ ayant pu apparaître après application d'une substitution sont remplacés par ces termes fermés de toutes les façons possibles.

Exemple 4 Considérons les deux sortes `int` et `s`, où `int` représente la sorte entière, et les symboles a, b : `int`, c : `s` et p : `int` $\times$ `s` $\rightarrow$ `bool`. Soit S l'ensemble de clauses suivant :

$$S \;=\; \{x \not\leq a \vee p(x,y),\, u \not\leq b \vee \neg p(u,c)\}\,,$$

où x et u sont des variables de sorte entière et y est une variable de sorte `s`. Si `int` est la sorte de base et `s` est la sorte englobante, alors S est un ensemble hiérarchique, et la procédure décrite réalise les opérations suivantes :

1. La clause C^N est extraite de chaque clause de S, on obtient ainsi l'ensemble $\{p(x,y),\, \neg p(u,c)\}$. Toutes les variables entières dans cet ensemble sont remplacées par l'élément $\bullet$, on obtient ainsi l'ensemble $S_N = \{p(\bullet,y),\, \neg p(\bullet,c)\}$.
2. La procédure d'instanciation Θ_N appliquée à S_N instancie la variable y par c et engendre donc l'ensemble $\{p(\bullet,c),\, \neg p(\bullet,c)\}$.
3. La substitution $y \mapsto c$ est appliquée aux clauses de S, on obtient donc l'ensemble de clauses S_B suivant :

   $$\{x \not\leq a \vee p(x,c),\, u \not\leq b \vee \neg p(u,c)\}\,,$$

   et cet ensemble ne contient que des variables de sorte entière.
4. La procédure d'instanciation pour les termes de sorte entière appliquée à l'ensemble $\{x \not\leq a,\, u \not\leq b\}$ instancie les variables par les constantes a et b.
5. Les variables de sorte entière de l'ensemble S_B sont donc instanciées de toutes les façons possibles par les constantes a et b ; on engendre ainsi après simplification l'ensemble de clauses fermées

   $$S' \;=\; \{p(a,c),\, b \not\leq a \vee p(b,c),\, a \not\leq b \vee \neg p(a,c), \neg p(b,c)\}\,,$$

   dont la satisfaisabilité peut être vérifiée par n'importe quel solveur SMT capable de traiter l'arithmétique et la logique propositionnelle. ♣

Nous avons imposé des conditions sur les procédures θ_N et θ_B afin de garantir que la procédure résultant de l'algorithme ci-dessus est complète. Intuitivement, ces conditions imposent que Θ_N soit complète sur les ensembles dont les termes d'une sorte de base sont fermés, que cette procédure soit indépendante des représentations des termes de sortes de base, et que le fait d'ajouter des clauses à un ensemble ne cause pas la génération d'un

nombre inférieur d'instances. La procédure Θ_B doit être complète à la base, et doit engendrer les mêmes instances pour un ensemble de clauses et le même ensemble auquel ont été ajoutées des clauses obtenues à partir de celles de départ en remplaçant des variables par des variables. Nous avons démontré que quand Θ_N et Θ_B vérifient ces conditions, la procédure $\Theta_N[\Theta_B]$ est complète.

4.5 Discussion

Nos travaux sur les procédures d'instanciation ont conduit à la réalisation d'une procédure générique qui est complète pour de nombreuses classes d'ensembles de clauses de la forme $\mathcal{T} \cup S$ et pour leur combinaison. Nous avons veillé à rendre cette procédure la plus efficace possible en restreignant l'ensemble des instanciations à appliquer aux différentes clauses. Nous avons également conçu une procédure spécifique applicable à l'arithmétique de Presburger qui était plus efficace que l'état de l'art, et défini à chaque fois des critères syntaxiques qui, s'ils sont vérifiés par l'ensemble de clauses fourni en entrée, garantissent la complétude de la procédure d'instanciation considérée. Ces différents résultats ont montré que les techniques que nous avons mis au point peuvent être utilisées dans de nombreux contextes. Ainsi, notre approche permet de résoudre de nombreux problèmes à l'aide de solveurs SMT capables de traiter quelques théories de base, comme l'arithmétique de Presburger : les axiomes des autres théories étant instanciés, il n'est pas nécessaire d'employer de techniques spécifiques pour les traiter. Les premiers tests d'intégration dans des solveurs ont été encourageants et mériteraient d'être poussés plus avant.

Chapitre 5

Démonstrations par induction

5.1 Présentation

L'induction est une façon courante de démontrer des théorèmes mathématiques. Par exemple, beaucoup de résultats simples de l'arithmétique, comme la commutativité de l'addition, se prouvent par récurrence simple, cette dernière pouvant être considérée comme une forme simplifiée du principe d'induction noethérienne (ou bien fondée). Ce type de preuve apparaît également très fréquemment en informatique où il est nécessaire de faire appel à l'induction pour raisonner sur des programmes contenant des itérations ou bien des structures de données définies de façon récursive, comme les listes ou les arbres. Dans le cas général, si E est un ensemble muni d'une relation d'ordre bien fondée [1] $\prec$, pour montrer que tous les éléments de E vérifient une propriété P, il suffit, pour un x fixé, de prouver que si tous les éléments $y \prec x$ vérifient P, alors x vérifie également P. D'un point de vue formel, ce principe d'induction peut être représenté par le schéma suivant :

$$\frac{\forall x.\, (\forall y.\, y \prec x \Rightarrow P(y)) \Rightarrow P(x)}{\forall x.\, P(x)}$$

P étant la conjecture à prouver. Dans le cas des entiers, il est plus fréquent d'employer le schéma d'induction suivant :

$$\frac{P(0) \quad \forall n : \mathbb{N}.\, P(n) \Rightarrow P(n+1)}{\forall n : \mathbb{N}.\, P(n)}$$

Ce principe est très puissant et il arrive fréquemment que des résultats qui semblent difficiles à prouver admettent des preuves simples et élégantes, basées sur un choix judicieux d'un ordre bien fondé et d'une conjecture pour l'induction.

1. Intuitivement, il ne peut pas exister pour cette relation d'ordre de suite infinie strictement décroissante d'éléments de E.

Un raisonnement par induction vise à démontrer une propriété universelle sur une variable qui doit être interprétée sur un ensemble inductif. Formellement, les propriétés démontrables par induction à partir d'une théorie $\mathcal{T}$ sont appelées les *conséquences inductives de* $\mathcal{T}$. Ce sont des formules qui doivent être satisfaites par tous les modèles de Herbrand de $\mathcal{T}$, mais il est possible que d'autres modèles de $\mathcal{T}$ ne soient pas des modèles de ces formules. Prenons par exemple une sorte `nat` munie des constructeurs $a :\rightarrow \texttt{nat}$ et $\mathrm{s} : \texttt{nat} \rightarrow \texttt{nat}$, et considéons la constante $n : \texttt{nat}$ ainsi que l'ensemble de clauses $S \stackrel{\text{def}}{=} \{p(a), \neg p(x) \vee p(\mathrm{s}(x)), \neg p(n)\}$. Pour $k \in \mathbb{N}$ fixé, il est clair que $S \models n \not\simeq \mathrm{s}^k(a)$, donc, en posant $U \stackrel{\text{def}}{=} \left\{ n \not\simeq \mathrm{s}^i(a) \;\middle|\; i \in \mathbb{N} \right\}$, on en déduit que $S \models U$. L'ensemble S est satisfaisable, un modèle en est l'interprétation I de domaine $\{0, 1, 2\}$, telle que $a^I = 0$, $n^I = 1$, et les fonctions s^I et p^I sont respectivement définies par :

$$\mathrm{s}^I : \left\{ \begin{array}{lcl} \mathrm{s}^I(0) & = & 2 \\ \mathrm{s}^I(1) & = & 1 \\ \mathrm{s}^I(2) & = & 2 \end{array} \right. \qquad p^I : \left\{ \begin{array}{lcl} p^I(0) & = & \top \\ p^I(1) & = & \bot \\ p^I(2) & = & \top \end{array} \right.$$

Cependant, S est insatisfaisable s'il est imposé que n soit interprété comme un terme de la forme $\mathrm{s}^k(0)$. En général, les conséquences inductives de $\mathcal{T}$ contiennent donc strictement les conséquences logiques de $\mathcal{T}$. Nous nous intéressons ici à l'automatisation du raisonnement par induction dans le cadre de la logique du premier ordre. Ceci est une tâche difficile, et une conséquence du théorème d'incomplétude de Gödel est qu'il n'est pas possible de construire un système d'inférence permettant de déterminer *toutes* les conséquences inductives d'une théorie [47].

Il est possible de rechercher des théorèmes inductifs en relaxant le problème d'origine, et en n'imposant plus que la variable d'induction soit définie sur un domaine inductif. Si la formule relaxée est insatisfaisable, alors celle d'origine l'est également, mais il est alors impossible de conclure quand la formule relaxée est satisfaisable. La recherche de théorèmes inductifs nécessite donc la mise au point de techniques spécifiques pour raisonner par induction. Le premier outil intégrant le raisonnement par induction et capable de démontrer des résultats non-triviaux est Nqthm [42], le précurseur de ACL2 [112]. De nouveaux outils, basés sur d'autres techniques ont été mis en place, comme par exemple RRL [111], SPIKE [37], INKA [26] ou Oyster/CLAM [49]. Ces outils reposent principalement sur deux approches :

L'induction explicite. Le principe de cette approche est d'utiliser des instances de la règle d'inférence représentant l'induction bien fondée dans la construction de preuves de théorèmes inductifs. L'essentiel des travaux dans cette branche portent sur la façon trouver automatiquement la bonne instance de la règle à utiliser. Ceci passe par la mise en place d'heuristiques, généralement basées sur l'*analyse de la récursion* [47, p. 21] pour la détection de règles d'inférence et la façon de contrôler

l'application de la règle d'induction [48], la découverte automatique de lemmes [110] ou la généralisation des hypothèses d'induction [3].

L'induction implicite. Cette approche est également appelée approche de la *preuve par consistance*. Parmi les techniques explorées, citons l'*induction sans induction* (*inductionless induction* [54]), dont le principe est de réduire le problème de la validité à un problème de satisfaisabilité. Cette technique a été mise au point dans le cas de la recherche de théorèmes inductifs d'un ensemble de clauses S satisfaisable, et uniquement constitué de clauses de Horn, quand le théorème à prouver est une clause C qui est positive. Dans ce cas-là, S admet un unique modèle de Herbrand minimal $\mathcal{H}$, et si $\mathcal{A}$ est une axiomatisation de $\mathcal{H}$ [2] alors C est une conséquence inductive de S si et seulement si $S \cup \mathcal{A} \cup \{C\}$ est satisfaisable. Cette approche a été généralisée au cas où S peut être un ensemble de clauses non-Horn, et C une clause quelconque dans [55]. Une autre approche, présentée dans [38], est basée sur la construction automatique d'ensembles tests, et consiste à prouver une conjecture inductive en combinant des opérations d'instanciations en utilisant des éléments de l'ensemble test, et de simplifications en utilisant les axiomes considérés, les conjectures déjà prouvées et des instances plus petites de la conjecture à prouver. Cette approche a été étendue pour traiter l'analyse par cas dans les théories conditionnelles [39], ou les opérateurs associatifs et commutatifs [24], puis généralisée au cas où, par exemple, les systèmes de réécriture considérés contiennent des équations qu'il est impossible d'orienter, comme la commutativité [135].

Plus récemment, de nouvelles techniques ont été mises au point pour prouver des propriétés inductives. Elles sont basées sur le principe de la *descente infinie*. Le principe de la descente infinie, attribué à Pierre de Fermat (mais employée depuis l'Antiquité), stipule que si, à chaque fois qu'une propriété n'est pas vérifiée sur un élément d'un ensemble muni d'une relation bien fondée $\prec$, elle n'est pas vérifiée sur un élément de l'ensemble strictement inférieur au premier, alors aucun élément de l'ensemble ne peut vérifier cette propriété. Les techniques mises au point peuvent être vues comme des variantes des *preuves cycliques* [45]. L'idée sous-jacente aux preuves cycliques est la suivante : Représentons une preuve par un arbre dont chaque sommet est étiqueté par un séquent s, auquel est associé une règle r, et dont les sommets fils sont étiquetés par $p_1, \ldots, p_m$ tels que $\dfrac{p_1 \quad \cdots \quad p_m}{s}$ est une instance de la règle r. Des cycles peuvent être identifiés dans un tel arbre en associant à des feuilles de cet arbre un sommet antérieur tel que ces deux sommets sont étiquetés par le même séquent à renommage près. Intuitive-

2. En toute rigueur, il est souvent impossible d'avoir une telle axiomatisation. L'ensemble $\mathcal{A}$ est alors un ensemble récursif de clauses universelles tel que $\mathcal{H}$ est l'unique modèle de Herbrand de $S \cup \mathcal{A}$.

ment, de tels cycles permettent de représenter un arbre de preuve avec des branches infinies, contenant un séquent qui se répète infiniment, ce qui signifie qu'une définition inductive a été développée une infinité de fois. Sous certaines conditions, le principe de la descente infinie s'applique et permet de déduire que le graphe avec cycle représente une démonstration valide. Cette technique de preuve semble aussi générale que les techniques standard de preuve par induction [46], et le démonstrateur QuodLibet [12, 177] est fondé sur ce principe. Il faut noter que, à notre connaissance, il n'existe pas de résultat de complétude dans le cas général concernant cette méthode.

Sur la schématisation

La notion de schéma est fréquemment utilisée en mathématiques. Elle permet typiquement d'obtenir une représentation compacte d'un ensemble éventuellement infini d'objets structurellement identiques, et son utilisation simplifie souvent les notations et en facilite la lecture. Un exemple de schéma est l'inégalité de Cauchy-Schwarz dans le cas réel :

$$\left(\sum_{i=1}^{n} x_i y_i\right)^2 \leq \left(\sum_{i=1}^{n} x_i^2\right)\left(\sum_{i=1}^{n} y_i^2\right).$$

En démonstration automatique, les schémas peuvent être utilisés à trois niveaux :

Au niveau des termes. Un schéma de termes permet de représenter un ensemble de termes similaires, construits à partir d'une base et d'un contexte inductif. Par exemple, $f(a, g(\circ))^n.a$ est un schéma de termes, et l'ensemble de termes représenté par ce schéma est obtenu en instanciant n ; ici, cet ensemble est $\{a, f(a, g(a)), f(a, g(f(a, g(a)))), \ldots\}$. Certaines classes de schémas de termes admettent des problèmes d'unification décidables, ce qui rend l'utilisation de tels schémas possible dans des systèmes d'inférence pour les rendre plus efficaces. Parmi les études sur les classes de schémas admettant des problèmes d'unification efficaces, on peut citer [51, 52, 156, 103].

Au niveau des formules. De tels schémas sont typiquement paramétrés par un ou plusieurs entiers, chaque instanciation de ces paramètres produisant une formule logique standard. Par exemple,

$$p_0 \wedge \bigwedge_{i=0}^{n-1} (p_i \Rightarrow p_{i+1}) \wedge \neg p_n$$

est un schéma de paramètre n, et chacune des formules obtenues en instanciant n par un entier est insatisfaisable. Le problème de satisfaisabilité sur des classes de telles formules est très fréquemment indécidable, même pour des classes en apparence simples de ces schémas

(voir par exemple [8]). Des systèmes d'inférence, basés sur la méthode des Tableaux et l'algorithme DPLL ont néanmoins été mis au point pour certaines classes de schémas [5, 6, 7].

Au niveau des preuves. Partant du principe qu'une preuve par induction peut être considérée comme représentant une infinité de preuves par déduction, il est possible de représenter une preuve par induction par un schéma de preuves déductives. Cette technique a été employée dans [14] pour l'analyse de la preuve topologique de Fürstenberg de l'infinité des nombres premiers, puis étendue dans [68].

5.2 Une généralisation de schémas de formules

De nombreux travaux sur les schémas de formules se restreignent au cas où les paramètres de ces schémas sont des entiers. Par exemple, les travaux dans [5, 7] considèrent le cas de schémas de formules propositionnelles paramétrées par des entiers, démontrent que sans des restrictions assez fortes, le problème de satisfaisabilité est indécidable et identifient des classes de schémas pour lesquelles ce problème est décidable. Les schémas identifiés permettent par exemple de modéliser des propriétés de circuits électroniques paramétrés par des entiers [100].

Nous avons étudié dans [73, 74] les schémas dans un cadre plus général que [5, 7], en relâchant les contraintes suivantes :

- Les paramètres des schémas ne sont pas nécessairement des entiers, mais des structures algébriques arbitraires telles que les listes ou les arbres.
- Les formules logiques obtenues en instanciant les paramètres par des valeurs concrètes ne sont pas nécessairement des formules de la logique propositionnelle, mais des formules de la logique du premier ordre, contenant éventuellement des symboles interprétés[3].

Les schémas de formules considérés sont définis sur un *langage de base*, et paramétrés par des structures inductives quelconques. Un intérêt majeur de cette approche est qu'elle permet une séparation claire entre le raisonnement sur les schémas et celui sur les formules de base, dont le traitement peut être délégué à des outils spécifiques, utilisés comme des boîtes noires. Une telle séparation n'était pas possible dans les approches de [5, 7], qui nécessitaient un entrelacement des deux types de raisonnements.

Avec cette nouvelle approche, les schémas sont représentés en sélectionnant un sous-ensemble de sortes, appelées les *sortes inductives*, qui sont utilisées pour la schématisation des formules, et en employant des *constructeurs*, dont les codomaines sont de sortes inductives, pour définir les domaines de ces sortes inductives. Les formules considérées sont construites à partir

3. D'autres travaux portant sur des formules plus expressives que celles de la logique propositionnelle ont également été menés dans [107].

d'atomes logiques standard et d'*atomes définis*, qui sont des éléments de la forme d_t, où d est un *symbole défini* et t un terme de sorte inductive. Ces atomes définis permettent de représenter des schémas de formules grâce à un système de réécriture convergent $\mathfrak{R}$. De nombreuses propriétés peuvent être représentées naturellement dans ce formalisme. Par exemple étant donnée une propriété p, nous pouvons représenter le fait qu'il existe dans un graphe dirigé acyclique (DAG) un chemin de la racine à une feuille dont tous les sommets sont étiquetés par des éléments vérifiant p. Nous considérons pour cela deux sortes `dag` et `elem`, pour respectivement représenter les sommets du DAG et les éléments étiquetant ses sommets. La sorte `dag` est inductive et admet deux constructeurs, $\bot :\rightarrow \texttt{dag}$ pour représenter une feuille et $\textsc{c} : \texttt{dag} \times \texttt{dag} \times \texttt{dag} \rightarrow \texttt{dag}$, où $\textsc{c}(n, l, r)$ dénote le DAG de racine n, fils gauche l et fils droit r. Le premier paramètre de $\textsc{c}$ permet notamment de garantir que deux sommets distincts peuvent avoir les mêmes descendants. Soit $\delta : \texttt{dag} \rightarrow \texttt{elem}$ une fonction qui associe à chaque sommet son étiquette. Etant donné un symbole défini E, l'existence d'un chemin dont toutes les étiquettes vérifient la propriété p s'exprime par la formule E_g, où g est un paramètre de sorte `dag`, avec le système de réécriture $\mathfrak{R}$ constitué des deux règles suivantes :

$$E_\bot \longrightarrow \top \qquad E_{\textsc{c}(n,l,r)} \longrightarrow (E_l \vee E_r) \wedge p(\delta(\textsc{c}(n, l, r))).$$

Nous avons mis au point une procédure basée sur la méthode des tableaux pour tester la satisfaisabilité de schémas de formules, éventuellement modulo une théorie. Les règles d'inférence de cette procédure appartiennent principalement à quatre catégories :

- les règles de décomposition, qui permettent de transformer une formule en conjonction ou disjonction de formules de base et littéraux définis ;
- les règles de dépliage, qui permettent de réécrire les atomes définis à l'aide du système de réécriture $\mathfrak{R}$;
- les règles équationnelles, pour raisonner sur les équations et diséquations ;
- les règles d'instanciation différée, qui permettent de considérer le cas où un paramètre est obtenu à partir de l'application d'un constructeur.

Une dernière règle permet de détecter un cycle le long d'une branche du tableau. Plus précisément, un cycle est détecté au niveau d'un sommet étiqueté par une formule Φ s'il existe un autre sommet étiqueté par une formule Ψ telle que $\Phi \models \Psi$. Les conditions d'application de la règle garantissent que la branche peut être fermée sans nuire à la correction de la procédure. Cette règle est essentielle pour assurer la terminaison de la procédure. Nous avons également défini des restrictions sur les schémas de formules et les interprétations considérées pour garantir la complétude du calcul. Intuitivement, les schémas de formules *admissibles* imposent des restrictions sur la forme des formules de base des schémas en question (les formules de base sont les

formules ne contenant aucun symbole défini), et les interprétations *schématisables* sont compatibles avec le système de réécriture $\mathfrak{R}$ (si $s \to_{\mathfrak{R}}$, alors s et t sont nécessairement interprétés de la même façon), et imposent que les éléments d'un domaine inductif soient obtenus à partir des constructeurs uniquement. Sous ces restrictions, nous avons prouvé que notre procédure termine et est complète : pour tester la satisfaisabilité d'un schéma de formule, il suffit de tester la satisfaisabilité des formules étiquetant les feuilles non-fermées de l'arbre construit. Autrement dit, le test de satisfaisabilité d'un schéma de formules se réduit au test de satisfaisabilité d'une disjonction *finie* de formules de base. De ces résultats on déduit que si le problème de satisfaisabilité est décidable (respectivement semi-décidable) pour la classe des formules de base considérées, alors il l'est également sur toute la classe des schémas de formules admissibles associée.

5.3 Une adaptation du calcul de Résolution pour les schémas de formules du premier ordre

Une majorité de travaux portant sur la détection de cycles pour l'automatisation des preuves par induction sont basés sur des calculs générant des preuves ayant des structures d'arbres, comme le calcul des séquents ou la méthode des tableaux. Les démonstrateurs automatiques modernes étant basés sur la saturation, nous avons étudié de quelle façon des techniques de détection de cycles peuvent être adaptées à de tels calculs pour prouver des théorèmes inductifs.

Nous avons défini dans [9] une adaptation de la Résolution à des schémas de formules propositionnelles après avoir encodé ces schémas de la façon suivante : nous considérons des ensembles de clauses, construites sur des variables propositionnelles indicées, des variables indicées et des paramètres. Chaque atome apparaissant dans une clause est un *atome équationnel*, de la forme $n \approx t$, ou bien un *atome indicé*, de la forme p_t, où n est un paramètre, p une variable propositionnelle indicée, et le terme t est de la forme $\mathrm{s}^k(u)$, le terme u étant une variable indicée ou bien la constante 0.

Exemple 5 Reprenons le schéma de formules présenté dans l'introduction :

$$p_0 \wedge \bigwedge_{i=0}^{n-1} (p_i \Rightarrow p_{i+1}) \wedge \neg p_n.$$

Dans ce formalisme, ce schéma peut être représenté par l'ensemble de clauses suivant :

$$(\star) \qquad \left\{p_0,\ \neg p_x \vee p_{\mathrm{s}(x)},\ n \not\approx y \vee \neg p_y\right\}. \qquad \clubsuit$$

Les règles de base du calcul que nous avons défini sont celles de la Résolution (voir Figure 5.1). Dans le cas où aucune des clauses dans l'ensemble

Résolution $\dfrac{p_u \vee C \qquad \neg p_v \vee D}{(C \vee D)\sigma}$

où :
$\sigma = \mathrm{mgu}(u, v)$, $p_u\sigma$ et $\neg p_v\sigma$ sont respectivement sélectionnés dans $(p_u \vee C)\sigma$ et $(p_v \vee D)\sigma$.

Factorisation $\dfrac{p_u \vee p_v \vee C}{(p_u \vee C)\sigma}$ $\qquad$ $\dfrac{\neg p_u \vee \neg p_v \vee C}{(\neg p_u \vee C)\sigma}$

où :
$\sigma = \mathrm{mgu}(u, v)$, $p_u\sigma$ est sélectionné dans $(p_u \vee p_v \vee C)\sigma$ et $\neg p_u\sigma$ est sélectionné dans $(\neg p_u \vee \neg p_v \vee C)\sigma$.

FIGURE 5.1 – Adaptation du calcul de Résolution

fourni en entrée ne contient pas d'atome équationnel, ce calcul correspond exactement à la Résolution standard, et il est donc garanti que ces règles appliquées à un ensemble instatisfaisable de clauses standard engendre la clause vide. Nous avons montré que, quand l'ensemble de clauses fourni en entrée est insatisfaisable et que certaines clauses contiennent des littéraux équationnels, ce calcul engendre un ensemble de clauses ne contenant aucun littéral indicé qui est insatisfaisable. Il se peut néanmoins que cet ensemble soit infini.

Exemple 6 Le calcul appliqué à l'ensemble de clauses ($\star$) engendre les clauses suivantes :

1	$n \not\approx x \vee \neg p_x$	(hyp)
2	$\neg p_y \vee p_{\mathrm{s}(y)}$	(hyp)
3	p_0	(hyp)
4	$n \not\approx \mathrm{s}(y) \vee \neg p_y$	(Résolution $1, 2$, $x \mapsto \mathrm{s}(y)$)
$4'$	$n \not\approx \mathrm{s}(y') \vee \neg p_{y'}$	(renommage, 4)
5	$n \not\approx 0$	(Résolution $1, 3$, $x \mapsto 0$)
6	$n \not\approx \mathrm{s}(0)$	(Résolution $4, 3$, $y \mapsto 0$)
7	$n \not\approx \mathrm{s}(\mathrm{s}(y)) \vee \neg p_y$	(Résolution $4', 2$, $y' \mapsto \mathrm{s}(y)$)
8	$n \not\approx \mathrm{s}(\mathrm{s}(0))$	(Résolution $7, 3$, $y \mapsto 0$)
	$\vdots$	

L'ensemble saturé engendré par ce calcul contient l'ensemble de clauses unitaires $\left\{ n \not\approx s^k(0) \;\middle|\; k \in \mathbb{N} \right\}$ qui est insatisfaisable, mais clairement, le calcul ne termine pas. ♣

Nous avons défini la classe des *clauses normalisées*[4], auxquelles peuvent être associées des niveaux, qui sont tels que chaque règle du calcul engendre

4. Même si cette classe est syntaxiquement restreinte, elle reste expressive, et nous avons montré comment associer à des clauses ne vérifiant pas les conditions de normalisation des clauses équivalentes qui sont normalisées (voir [9, Sec. 5]).

une conséquence dont le niveau est au moins égal à celui des prémisses. Etant donné un ensemble de clauses normalisées S, l'ensemble des clauses S' engendrées par le calcul à partir de S et auxquelles les littéraux équationnels ont été retirés est fini. Nous avons utilisé ce résultat pour démontrer qu'il est alors possible d'identifier deux niveaux i et j tels que S et $S \cup \{n \not\approx \mathrm{s}^{i+j}(x)\}$ sont équisatisfaisables. Intuitivement, les niveaux i et j sont tels que les ensembles S'_i et S'_{i+j}, contenant respectivement les clauses non-fermées de S' de niveau i et $i+j$, sont équivalentes, si n est remplacé par $n-j$ dans les clauses de S'_i ; une telle équivalence traduit l'existence d'un cycle dans la construction de la réfutation de S. Ainsi, le calcul obtenu en ajoutant la règle d'inférence qui, quand les niveaux i et j sont identifiés, ajoute à l'ensemble de clauses considéré la clause unitaire $n \not\approx s^{i+j}(x)$ (appelée *clause de coupure*), est correct, et nous avons démontré que le calcul résultant termine. Le résultat de terminaison prouve que le problème de satisfaisabilité est décidable pour la classe des ensembles de clauses normalisées.

Exemple 7 Considérons les clauses engendrées dans l'Exemple 6, et soient $i = 2$ et $j = 1$. L'unique clause de niveau 2 engendrée par le calcul est $C \stackrel{\text{def}}{=} n \not\approx \mathrm{s}(y) \vee p_y$, et l'unique clause non-fermée de niveau 3 est $C' \stackrel{\text{def}}{=} n \not\approx \mathrm{s}(\mathrm{s}(y)) \vee p_y$. Le remplacement de n par $n-j$ dans C consiste à remplacer l'atome équationnel de C par $n \not\approx \mathrm{s}(\mathrm{s}(y))$, et la clause résultante est égale à C'. Ceci signifie que la clause de coupure $n \not\approx \mathrm{s}(\mathrm{s}(\mathrm{s}(x)))$ peut être ajoutée à l'ensemble de clauses engendrées, et le calcul termine alors après avoir engendré la clause vide. ♣

Un avantage du calcul défini est qu'il s'étend naturellement à des schémas de clauses du premier ordre, construites à partir de paramètres, termes du premier ordre et prédicats indicés. La seule contrainte supplémentaire imposée est que les termes apparaissant dans les prédicats indicés et ceux apparaissant dans les indices sont disjoints : des atomes tels que $Q(0)_x$ ne sont pas autorisés. La règle d'inférence ajoutant une clause de coupure à un ensemble de clauses reste valable dans ce cadre, mais, comme il est possible d'avoir une infinité de clauses normalisées d'un niveau donné, ceci ne garantit pas la terminaison du calcul.

5.4 Discussion

A notre connaissance, il n'y avait pas de travaux de recherche se rapprochant des nôtres sur les schémas de formules généralisés, et plus particulièrement sur les résultats de décidabilité qui y sont démontrés. Ces schémas peuvent être employés sur plusieurs structures de données comme les entiers, les listes et les arbres, et permettent de traiter de nombreuses classes de formules. Il serait intéressant d'implémenter la procédure que nous avons définie, pour en vérifier expérimentalement l'efficacité.

Nous estimons qu'il serait intéressant d'étendre nos travaux sur l'intégration du raisonnement par induction au calcul de Résolution. Une extension naturelle est l'intégration du raisonnement par induction au calcul de Superposition. C'est une piste que nous explorons en ce moment et qui est décrite au Chapitre 7.

Chapitre 6

Raisonnement par abduction

L'*abduction* est une des trois formes de raisonnements distinguées par Peirce [102], avec la déduction (la dérivation de conclusions en appliquant des règles d'inférences à des hypothèses) et l'induction (la construction de règles d'inférences à partir d'hypothèses et de conclusions). C'est un processus synthétique qui, à l'aide de règles d'inférences, construit des hypothèses manquantes à partir d'une base de connaissances et d'observations inexpliquées. Autrement dit, le raisonnement abductif consiste à rechercher les explications plausibles de faits observés. L'abduction a été beaucoup étudiée en Intelligence Artificielle, dans le cadre du raisonnement en présence d'une information incomplète. Ce mode de raisonnement a par exemple été employé pour le diagnostic médical ou basé sur le modèle [145, 101, 152], la reconnaissance visuelle [58, 146], la compréhension de textes [106] ou la planification [132].

D'un point de vue formel, étant donnée une théorie $\mathcal{T}$ représentant un ensemble de connaissances et une observation O que la base de connaissances ne permet pas d'expliquer ($\mathcal{T} \not\models O$), le raisonnement abductif consiste à rechercher des causes H telles que $\mathcal{T} \wedge H \models O$ (H permet d'expliquer O) et $\mathcal{T} \wedge H$ est satisfaisable (H et $\mathcal{T}$ ne sont pas contradictoires). En général, des restrictions additionnelles sont imposées à H pour éviter, par exemple, d'expliquer l'observation O en prenant $H \stackrel{\text{def}}{=} O$; l'ensemble des causes possibles auxquelles doit appartenir H est appelé l'ensemble des *abducibles*. Il est également courant d'imposer que H soit une cause la plus générale possible, pour éviter par exemple de prendre pour H une cause qui est elle-même explicable par d'autres causes.

Nous nous sommes intéressés au raisonnement abductif dans le cadre de la correction automatique d'erreurs. Un exemple d'application en est la détection de bugs dans les programmes. Pour prouver qu'un programme se comporte comme prévu, il est possible de le modéliser par une formule $\mathcal{T}$, le comportement qu'il est supposé observer par une formule O, puis de vérifier que $\mathcal{T} \models O$. Un démonstrateur complet pour la réfutation peut être employé

pour cette tâche puisque, par dualité, il suffit de vérifier que $\mathcal{T} \wedge \neg O$ est insatisfaisable. Quand c'est le cas, le démonstrateur peut engendrer une preuve démontrant que le programme se comporte effectivement comme prévu. En revanche, quand $\mathcal{T} \wedge \neg O$ est satisfaisable, ceci signifie que le programme comporte une erreur, et les modèles de cette formule représentent des traces d'exécution du programme menant à un comportement indésiré ; une analyse de ces modèles permettrait alors à l'utilisateur de corriger le programme. De tels modèles peuvent être construits automatiquement en utilisant, par exemple, les techniques décrites dans [50], et de nombreux outils dont plusieurs solveurs SMT sont capables de telles constructions. Cette analyse de modèles peut néanmoins s'avérer longue et difficile car la formule satisfaisable admet généralement un grand nombre de modèles sur des domaines différents, et ces modèles peuvent être larges et complexes. Extraire de ces modèles une propriété générale qui explique le comportement indésirable n'est donc pas une tâche triviale. Nous conjecturons qu'une information pertinente à obtenir quand une formule est satisfaisable serait une propriété qui est *commune* à tous les modèles de cette dernière. Illustrons cette conjecture avec un exemple. Reprenons la théorie des tableaux, dont les axiomes ont été introduits dans la Figure 3.3, page 27, et cherchons à démontrer que l'ordre dans lequel des éléments sont insérés dans un tableau n'a aucune importance (voir la Figure 6.1). Cette propriété est valide si et seulement si la formule suivante est insatisfaisable dans la théorie des tableaux :

$$\text{select}(\text{store}(\text{store}(a, i, b), j, c), k) \not\simeq \text{select}(\text{store}(\text{store}(a, j, c), i, b), k).$$

Cette formule exprime le fait qu'il existe un indice k qui contient des valeurs distinctes dans le tableau obtenu en insérant dans le tableau a d'abord l'élément b à l'indice i puis l'élément c à l'indice j, et dans celui obtenu en insérant d'abord c à l'indice j puis b à l'indice i. Cette formule est satisfaisable, ce qui veut dire que l'ordre dans lequel sont insérés les éléments peut être important. Pour en comprendre la raison, il est possible d'analyser des modèles de cette formule. Le modèle construit par le solveur SMT Yices[1] pour cette formule est `(= b 1) (= c 3)(= i 2) (= k 2) (= j 2)`, et pour cette formule, ce modèle suffit à comprendre comment corriger ce problème.

Une façon plus simple de détecter et corriger le problème serait d'avoir un outil qui *explique* pourquoi cette formule est satisfaisable ; ici, un tel outil engendrerait la formule $i \simeq j \wedge b \not\simeq c$, ce qui signifie que les deux tableaux peuvent en effet être distincts à condition que les éléments b et c soient différents, et insérés à la même position dans le tableau. La négation de cette formule correspond à deux hypothèses qui, en étant ajoutées, rendent la formule insatisfaisable : $b \simeq c$ et $i \not\simeq j$. Ce qui signifie que, pour que l'ordre dans lequel des éléments sont ajoutés dans un tableau n'ait pas d'importance,

1. http://yices.csl.sri.com/

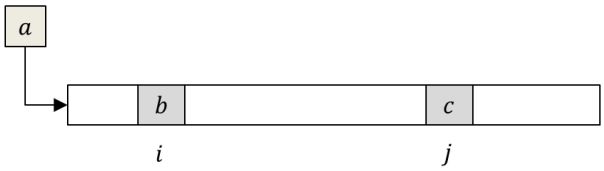

FIGURE 6.1 – Insertion dans le tableau a de l'élément b à l'indice i et de l'élément c à l'indice j.

il faut préciser que ces éléments sont identiques ($b \simeq c$), ou alors qu'ils sont insérés à des positions différentes ($i \not\simeq j$).

Les impliqués premiers

Une façon de rechercher les explications d'une observation est d'utiliser l'équivalence logique entre $\mathcal{T} \wedge H \models O$ et $\mathcal{T} \wedge \neg O \models \neg H$, et transformer ainsi un problème d'abduction en un problème de déduction : afin de rechercher les causes d'une observation O avec la base de connaissances $\mathcal{T}$, il est possible de rechercher les *conséquences* de $\mathcal{T} \wedge \neg O$. De telles conséquences sont appelées des *impliqués*, et les conséquences les plus générales des *impliqués premiers*. Formellement, un impliqué d'une formule F est une clause C qui n'est pas une tautologie, telle que $F \models C$. La clause C est un impliqué premier si de plus, à chaque fois que D est un impliqué de F tel que $D \models C$, on a $C \models D$. Nous avons utilisé ce procédé pour la recherche d'explications de formules satisfaisables car de nombreux travaux de recherche ont été menés auparavant sur la génération d'impliqués premiers, notamment dans le cadre de la logique propositionnelle, où le concept a été étudié par Quine [149]. Les premiers outils mis au point étaient basés sur une représentation par des *mintermes* de la formule considérée (voir par exemple [163]), mais étaient peu efficaces. Les outils ultérieurs capables d'engendrer des impliqués premiers en logique propositionnelle peuvent pour la plupart être classés en deux catégories :

- Les outils basés sur des méthodes de décomposition. Ces outils construisent les impliqués premiers d'une formule de façon récursive, en décomposant la formule considérée en sous-formules, et en combinant les impliqués obtenus pour ces sous-formules [56, 123, 124].
- Les outils basés sur la Résolution. Ces outils sont tous basés sur le même algorithme, qui consiste à chaque étape en la génération de résolvantes des éléments de l'ensemble de clauses considéré, d'où sont systématiquement retirées les clauses les moins générales. Le point fixe de ce processus est l'ensemble des impliqués premiers des clauses de départ. Les algorithmes emploient généralement la même stratégie pour engendrer de nouvelles résolvantes, en considérant les littéraux de façon

PiTison(S) =
// entrée : S l'ensemble de clauses de départ
// sortie : P l'ensemble des impliqués premiers
$P := S$
pour chaque variable propositionnelle p **faire**
tant que il existe C résolvante sur p de clauses de P **faire**
soit $P' = \{E \in P \mid E$ est moins générale que $C\}$ **dans**
$P := (P \setminus P') \cup \{C\}$
fin tant que
fin pour
retourner P

FIGURE 6.2 – Algorithme de Tison pour le calcul d'impliqués premiers

incrémentale, [169, 113, 108] (voir aussi la Figure 6.2), et diffèrent principalement par les structures de données compactes employées pour stocker les impliqués premiers, comme les tries [92, 62] ou les ZBDD [131, 160]. Nous avons choisi dans nos travaux d'adapter cette méthode plus uniforme à la logique équationnelle.

Un nouvel outil a récemment été mis au point [148], qui offre de bien meilleures performances que ceux présentés ci-dessus pour la génération d'impliqués premiers. L'approche employée par les auteurs est de transformer le problème d'origine en associant à chaque littéral l apparaissant dans la formule d'origine une variable propositionnelle x_l qui est utilisée pour indiquer si le littéral apparaît ou non dans un impliqué premier, et d'utiliser un solveur SAT pour énumérer, par raffinements successifs tous les impliqués premiers de la formule.

Il y a, à notre connaissance, beaucoup moins de travaux de recherche portant sur la génération d'impliqués premiers dans des logiques plus expressives que la logique propositionnelle. Certaines extensions ont été considérées en logique modale [25, 28], et la plupart des méthodes pour les extensions à la logique du premier ordre reposent sur la Résolution [114, 122] ou la méthode des Tableaux [125, 134].

Pour un ensemble de symboles abducibles $\mathcal{A}$ donné, nous appelons *$\mathcal{A}$-impliqués premiers* d'un ensemble S l'ensemble des impliqués premiers de S contenant uniquement des symboles de $\mathcal{A}$. Nous avons cherché à résoudre le problème suivant : étant donné un ensemble de clauses S et un ensemble de symboles abducibles $\mathcal{A}$, comment construire efficacement l'ensemble des $\mathcal{A}$-impliqués premiers de S ? Nous avons étudié la génération d'impliqués premiers d'ensembles de clauses dans lesquelles apparaît le prédicat d'égalité. Les outils existants permettent d'accomplir cette tâche : il est en effet garanti que le calcul de Résolution standard permet d'engendrer l'ensemble des impliqués recherchés, à condition d'ajouter à l'ensemble de clauses considéré

des instances des axiomes de l'égalité. Cette approche présente néanmoins des inconvénients car le nombre nécessaire d'instances des axiomes de l'égalité à ajouter devient rapidement important et tous les impliqués premiers qui sont uniquement des conséquences de ces axiomes doivent être filtrés a posteriori. Cette approche n'est donc pas efficace, et il n'est pas clair que l'algorithme résultant termine sur les classes d'ensembles de clauses qui peuvent par exemple apparaître dans les problèmes de vérification. C'est pourquoi nous avons abordé le problème en nous plaçant en logique équationnelle, dans laquelle il n'est pas nécessaire d'explicitement considérer les axiomes de l'égalité. Un choix naturel de système d'inférence a été le calcul de superposition, mais ce dernier ne peut pas être utilisé pour engendrer l'ensemble des $\mathcal{A}$-impliqués premiers d'un ensemble de clauses, mis à part dans certains cas particuliers (voir par exemple [171]). Ainsi, aucune règle d'inférence ne peut être appliquée à l'ensemble de clauses $\{f(a) \not\simeq f(b)\}$, bien que $a \not\simeq b$ soit un impliqué premier de cet ensemble. Nous avons donc exploré de quelles façons ce calcul pouvait être adapté pour engendrer tous les impliqués premiers d'un ensemble de clauses. Une grande partie de ces travaux ont été réalisés dans le cadre de la thèse de Sophie Tourret [170].

6.1 Symboles abducibles constants

Nous avons considéré le cas où les symboles abducibles considérés constituent un ensemble fini de termes fermés. Quitte à aplatir ces termes, ceci nous permet de supposer que l'ensemble $\mathcal{A}$ est constitué d'un ensemble de constantes, ce qui garantit que, à idempotence près, l'ensemble des $\mathcal{A}$-impliqués premiers d'un ensemble de clauses donné est fini.

6.1.1 Filtrage des symboles non-abducibles

Une première approche a consisté à étudier comment utiliser le calcul de superposition pour *filtrer* les symboles non-abducibles d'un ensemble de clauses :

Définition 8 Etant donnés des ensembles de clauses S et S', on dit que S' est un *$\mathcal{A}$-filtrage de S* si et seulement si :
- $S \models S'$.
- Les uniques symboles apparaissant dans S' sont des éléments de $\mathcal{A}$.
- Les $\mathcal{A}$-impliqués premiers de S sont exactement les $\mathcal{A}$-impliqués premiers de S'. ◊

Clairement, engendrer les $\mathcal{A}$-impliqués premiers de S en appliquant le calcul de Résolution standard à un $\mathcal{A}$-filtrage de S est plus efficace que d'appliquer ce calcul directement à S ; il est de plus garanti qu'un tel procédé termine, ce qui peut ne pas être le cas pour l'ensemble S. Nous avons défini dans [72]

une adaptation du calcul de Superposition qui, étant donné un ensemble de clauses S, permet d'engendrer un $\mathcal{A}$-filtrage de S.

L'adaptation proposée est basée sur un principe d'abstraction similaire à celui employé dans le calcul hiérarchique de [16], avec la différence notable que dans ce calcul, aucune contrainte n'empêche les symboles abducibles d'interagir avec les autres termes dans les clauses. Plus précisément, les termes considérés dans ce calcul sont construits sur un ensemble de variables de la forme $\mathcal{V} \uplus \mathcal{V}_\mathcal{A}$, les éléments de $\mathcal{V}$ étant des variables standard, et les éléments de $\mathcal{V}_\mathcal{A}$ des *variables abducibles*. Ces variables abducibles dénotent intuitivement des constantes abducibles, et permettent d'associer à des clauses standard des clauses équivalentes dont les littéraux contenant des symboles autres que des constantes ou variables abducibles, ne contiennent *aucune* constante abducible. De telles clauses sont appelées des *clauses abstraites*.

Exemple 9 Posons $\mathcal{A} = \{a, b, c\}$, $C = f(a, x) \simeq b \vee a \not\simeq d \vee g(c) \simeq g(f(y, c))$, et supposons que $\{x_a, x_b, x_c\} \subseteq \mathcal{V}_\mathcal{A}$. Alors la clause C est équivalente à la clause abstraite suivante :

$$x_a \not\simeq a \vee x_b \not\simeq b \vee x_c \not\simeq c \vee f(x_a, x) \simeq x_b \vee x_a \not\simeq d \vee g(x_c) \simeq g(f(y, x_c)).$$

♣

Intuitivement, ce principe d'abstraction permet d'employer le calcul de Superposition pour engendrer des $\mathcal{A}$-impliqués supplémentaires. Par exemple, dans le cas où $\mathcal{A} = \{a, b\}$, l'abstraction de la clause unitaire $f(a) \not\simeq f(b)$ est $x \not\simeq a \vee y \not\simeq b \vee f(x) \not\simeq f(y)$. La règle de Réflexion appliquée à cette clause produit $x \not\simeq a \vee x \not\simeq b$ et, en appliquant à nouveau cette règle, le calcul engendre $a \not\simeq b$, qui est bien un $\mathcal{A}$-impliqué de $f(a) \not\simeq f(b)$. Cependant, ce principe d'abstraction ne permet pas toujours d'engendrer tous les $\mathcal{A}$-impliqués premiers d'un ensemble de clauses, et il ne permet pas non plus d'obtenir un $\mathcal{A}$-filtrage de cet ensemble de clauses. Nous avons donc adapté le calcul de Superposition en imposant des restrictions sur les unificateurs autorisés pour appliquer les règles d'inférence, et sur les ordres de simplification utilisés pour comparer les termes et littéraux.

- Les restrictions imposées sur les unificateurs utilisés dans le calcul interdisent que les images des variables abducibles soient des éléments autres que des variables (abducibles ou standard), et que les images des variables standard contiennent des constantes abducibles. Ces restrictions permettent de garantir que la conclusion de chaque règle d'inférence est une clause abstraite et bloquent certaines inférences qui sont inutiles dans le processus de filtrage.
- Les restrictions sur les ordres de simplification considérés garantissent que l'ordre entre deux termes ne dépend pas des constantes abducibles qu'ils contiennent.

Après avoir défini une notion de redondance adaptée à ce calcul qui coïncide avec la notion de redondance standard pour des clauses standard, nous avons

démontré que si S est un ensemble saturé de clauses abstraites, alors

$$\mathrm{F}_{\mathcal{A}}(S) \stackrel{\text{def}}{=} \{C \in S \mid C \text{ contient uniquement des symboles de } \mathcal{A}\}$$

est un $\mathcal{A}$-filtrage de S. Nous avons également défini un critère garantissant la terminaison de ce calcul, ce qui nous a permis de prouver que, pour tout ensemble de constantes abducibles $\mathcal{A}$, le calcul termine en engendrant un $\mathcal{A}$-filtrage de tout ensemble de la forme $\mathcal{T} \cup S$, où S est un ensemble de clauses fermées et $\mathcal{T}$ est une des théories définies dans [11, 10, 34]. Nous obtenons donc une procédure qui termine et permet d'engendrer des $\mathcal{A}$-impliqués premiers (et donc des explications construites sur les symboles de $\mathcal{A}$) dans de nombreuses théories fréquemment utilisées dans la communauté SMT, en appliquant les étapes suivantes à l'ensemble de clauses S fourni en entrée :

1. Abstraction des clauses dans S pour obtenir un ensemble équivalent S', constitué de clauses abstraites.
2. Saturation de S' en utilisant l'adaptation du calcul de Superposition définie ci-dessus pour obtenir un ensemble saturé T.
3. Extraction de $\mathrm{F}_{\mathcal{A}}(T)$ de l'ensemble T.
4. Saturation de $\mathrm{F}_{\mathcal{A}}(T) \cup Eq$ par le calcul de Résolution standard, où Eq contient les instances par les éléments de $\mathcal{A}$ des axiomes de l'égalité.

L'ensemble saturé obtenu à l'étape 4 est exactement l'ensemble des $\mathcal{A}$-impliqués premiers de S.

6.1.2 Une génération efficace des $\mathcal{A}$-impliqués premiers

Définition du calcul $\mathcal{K}$

La procédure décrite ci-dessus permet de réduire le problème de la génération des $\mathcal{A}$-impliqués premiers d'un ensemble de clauses quelconque à la génération des $\mathcal{A}$-impliqués premiers d'un ensemble de clauses fermées ne contenant que des symboles de $\mathcal{A}$. La génération des $\mathcal{A}$-impliqués premiers pour de tels ensembles en se basant sur le calcul de Résolution et l'instanciation des axiomes de l'égalité demeure néanmoins peu efficace. Par exemple, il est possible d'appliquer la règle de Résolution avec pour prémisses deux instances de l'axiome de transitivité. Une telle inférence est clairement inutile, puisque la résolvante est une tautologie en logique équationnelle. Nous avons proposé dans [75, 76] un système d'inférence $\mathcal{K}$, qui intègre le raisonnement équationnel, pour engendrer les $\mathcal{A}$-impliqués premiers. Ce calcul est une adaptation du calcul standard de Paramodulation non-ordonnée, et est constitué des règles d'inférence de la Figure 6.3.

Ce calcul peut être interprété comme un calcul de Paramodulation conditionnelle. Par exemple les clauses unitaires $a \simeq b$ et $c \not\simeq d$ permettent d'engendrer la clause $a \not\simeq c \vee b \not\simeq d$, qui peut s'interpréter comme : *"si a et c sont*

Factorisation

$$\frac{a \simeq b \vee a' \simeq b' \vee C}{a \simeq b \vee a \not\simeq a' \vee b \not\simeq b' \vee C}$$

Paramodulation positive

$$\frac{a \simeq b \vee C \qquad a' \simeq c \vee D}{a \not\simeq a' \vee b \simeq c \vee C \vee D}$$

Multi-paramodulation négative

$$\frac{\bigvee_{i=1}^{n}(a_i \not\simeq b_i) \vee P_1 \qquad c \simeq d \vee P_2}{\bigvee_{i=1}^{n}(a_i \not\simeq c \vee d \not\simeq b_i) \vee P_1 \vee P_2}$$

FIGURE 6.3 – Règles d'inférence du calcul $\mathcal{K}$

égaux alors b et d sont distincts". Le calcul $\mathcal{K}$ admet également une règle de suppression de clauses redondantes : pour un ensemble de clauses $S \cup \{C\}$, s'il existe une clause $D \in S$ telle que $D \models C$, alors la clause C est supprimée de l'ensemble de clauses considéré. Ce critère de redondance est essentiel pour obtenir un calcul efficace et garantir que l'ensemble saturé obtenu par l'application des règles contient exactement l'ensemble des impliqués premiers recherché, sans qu'il soit nécessaire d'avoir recours à un post-traitement. Il explique pourquoi le calcul admet une règle de Multi-paramodulation négative et non pas une règle de Paramodulation négative : dans le cas contraire, le calcul n'aurait pas permis d'engendrer tous les impliqués premiers d'un ensemble de clauses en présence du critère de redondance.

Exemple 10 Soit $S \stackrel{\text{def}}{=} \{a \not\simeq c \vee a \not\simeq d,\ a \simeq b \vee b \not\simeq c \vee b \not\simeq d\}$, et considérons la clause $C \stackrel{\text{def}}{=} b \not\simeq c \vee b \not\simeq d$. Alors C est un impliqué premier de S, puisque $S \cup \{b \simeq c, b \simeq d\}$ est insatisfaisable. Cependant, si la règle de Multi-paramodulation négative est remplacée par une règle de Paramodulation négative dans $\mathcal{K}$, alors la clause C ne peut pas être engendrée à partir de S car toutes les clauses engendrées à partir de S sont redondantes. Par exemple, la clause engendrée par la règle de Paramodulation négative avec pour prémisses les deux éléments de S est $a \not\simeq a \vee b \not\simeq c \vee a \not\simeq d \vee b \not\simeq c \vee b \not\simeq d$; cette clause est équivalente à $a \not\simeq c \vee a \not\simeq d$ et est donc supprimée par le critère de redondance. La règle de Multi-paramodulation négative appliquée aux deux clauses de S, en prenant $a_1 = a_2 = a$, $b_1 = c$ et $b_2 = d$, engendre la clause $a \not\simeq a \vee b \not\simeq c \vee a \not\simeq a \vee b \not\simeq d \vee b \not\simeq c \vee b \not\simeq d$, qui est équivalente à C. ♣

Nous avons démontré que $\mathcal{K}$ est un calcul complet : si S est un ensemble de clauses saturé et C est un impliqué de S, alors il existe $D \in S$ tel que $D \models C$; un ensemble de clauses saturé par $\mathcal{K}$ est donc identique à son propre ensemble d'impliqués premiers.

Représentation et stockage des impliqués

L'efficacité de l'implémentation de $\mathcal{K}$ repose sur la façon dont sont représentés et stockés les impliqués engendrés. Ce problème de stockage a été largement étudié en logique propositionnelle, où des structures de données comme les tries [92] ou les ZBDD [130] sont utilisés dans les générateurs d'impliqués premiers les plus performants. Adapter ces structures de données à la logique équationnelle nécessite de résoudre plusieurs problèmes, même dans le cas où les clauses considérées ne contiennent que des constantes dans $\mathcal{A}$. Ainsi, en logique propositionnelle, tester une relation de conséquence logique entre deux clauses se réduit à effectuer un test d'inclusion, et deux clauses sont équivalentes si et seulement si elles sont égales (modulo idempotence et commutativité de l'opérateur de disjonction). Ce n'est plus le cas en logique équationnelle, où une clause peut être la conséquence logique d'une autre clause, sans avoir le moindre littéral en commun, et où plusieurs clauses peuvent être équivalentes tout en étant syntaxiquement distinctes. Nous avons défini une opération de projection sur les clauses, qui permet de vérifier syntaxiquement si une clause est la conséquence logique d'une autre clause, et une opération de normalisation, qui associe à deux clauses équivalentes la même forme normale. Ces opérations sont basées sur des relations d'équivalence qui peuvent être associées à des clauses et sur l'ordre $\prec_{\mathcal{A}}$ employé pour comparer les symboles abducibles entre eux. Intuitivement, à une clause C est associée une relation d'équivalence $\equiv_C$ telle que pour tous termes t et s, on a $t \equiv_C s$ si et seulement si $\neg C \models t \simeq s$. La projection d'une clause D sur C se construit à l'aide des plus petits éléments pour $\prec_{\mathcal{A}}$ des classes de congruence de $\equiv_C$. La forme normale $C_{\downarrow}$ de C est intuitivement la plus petite clause pour une extension de $\prec_{\mathcal{A}}$ qui est équivalente à C.

Exemple 11 Supposons que $\mathcal{A} = \{a, b, c, d, e\}$, où $a \prec_{\mathcal{A}} b \prec_{\mathcal{A}} c \prec_{\mathcal{A}} d \prec_{\mathcal{A}} e$ et considérons les clauses $C \stackrel{\text{def}}{=} a \not\simeq b \vee b \not\simeq c \vee a \simeq d \vee b \simeq e$, et $D \stackrel{\text{def}}{=} b \not\simeq d \vee c \simeq e$. Comme $\neg C \models a \simeq b \wedge b \simeq c$, les classes d'équivalence de la relation $\equiv_C$ sont $\{a, b, c\}$, $\{d\}$ et $\{e\}$, et les plus petits éléments de chaque classe pour $\prec_{\mathcal{A}}$ sont respectivement a, d et e. La projection de D sur C est la clause $a \not\simeq d \vee a \simeq e$ et la forme normale de C est $C_{\downarrow} = a \not\simeq b \vee a \not\simeq c \vee a \simeq d \vee a \simeq e$.♣

Ces opérations nous ont permis de définir les arbres clausaux, une adaptation des tries propositionnels, avec des algorithmes efficaces d'insertion et suppression de clauses pour stocker les impliqués engendrés par le calcul.

Le calcul a été implémenté [2] en OCaml. L'outil résultant, nommé `Kparam`, est basé sur l'algorithme "given clause" [127] (voir aussi l'Algorithme 1) : les clauses de l'ensemble considéré sont initialement stockées dans un ensemble W de clauses à traiter (*waiting set*) et sont extraites une par une et utilisées comme prémisses des règles d'inférence du calcul avant d'être stockées dans

2. http://lig-membres.imag.fr/tourret/index.php?&tab=3&slt=tools

Algorithme 1 : Algorithme *Given Clause*

entrée : un ensemble de clauses S
sortie : les impliqués premiers de S
$W := S$;
$P := \emptyset$;
tant que $W \neq \emptyset$ **faire**
  retirer C de W;
  si C *n'est pas redondant dans* P **alors**
    $R := \{D \in P \mid D \text{ est redondant dans } P \cup \{C\}\}$;
    $P := (P \cup \{C\}) \setminus R$;
    $N :=$
    $\{D \text{ engendré avec prémisses dans } P, \text{ dont au moins un est } C\}$;
    $W := W \cup N$;
retourner P

un ensemble de clauses traitées (*processed set*). Les clauses redondantes sont éliminées au fur et à mesure qu'elles sont détectées et les clauses nouvellement engendrées sont stockées dans l'ensemble W.

Nous avons effectué une étude expérimentale sur l'outil résultant, `Kparam`. Comme il n'y avait, à notre connaissance, aucun outil permettant d'engendrer tous les impliqués premiers en logique équationnelle, nous avons instancié les problèmes considérés en logique propositionnelle, et comparé `Kparam` avec `Zres`, qui était alors l'outil le plus efficace pour engendrer les impliqués premiers dans cette logique. Nous avons comparé les temps nécessaires aux deux outils pour engendrer les impliqués premiers, et n'avons pas inclus dans ce décompte le temps nécessaire au post-traitement des impliqués propositionnels obtenus pour, par exemple, supprimer ceux qui sont des tautologies en logique équationnelle. Les résultats expérimentaux ont montré que `Kparam` est plus rapide que `Zres` dans 65% des cas, et au moins deux fois plus rapide dans 54% des cas.

6.1.3 Exploitation des impliqués unitaires

Une analyse détaillée des cas dans lesquels l'outil `Kparam` est moins efficace que `Zres` nous a permis de constater qu'en général, ceci se produisait quand l'ensemble de clauses considéré admet des impliqués unitaires. Nous avons étudié de quelle façon exploiter de tels impliqués et nous sommes basés sur le fait que si une formule F admet un impliqué unitaire $a \simeq b$, alors $F \equiv F \wedge a \simeq b \equiv F[b/a] \wedge a \simeq b$, où $F[b/a]$ est la formule obtenue en remplaçant chaque occurrence de a par b dans F. Clairement, il est moins coûteux d'engendrer les impliqués premiers de $F[b/a]$ que ceux de F, et nous avons étudié comment exploiter la présence d'impliqués unitaires pour accélérer la

génération des impliqués premiers d'un ensemble de clauses donné.

Une première approche a consisté à mettre au point un calcul permettant de générer tous les impliqués premiers unitaires d'un ensemble de clauses, afin de simplifier ce dernier avant d'en engendrer les autres impliqués premiers. Nous avons montré (voir [170]) que le calcul de Paramodulation non-ordonnée permet d'engendrer ces impliqués unitaires ; plus précisément, si S est un ensemble saturé par la Paramodulation non-ordonnée et C est une clause unitaire telle que $S \models C$, alors $C \in S$. Les impliqués unitaires de S peuvent donc être obtenus en extrayant toutes les clauses unitaires de la saturation par la Paramodulation non-ordonnée de S. Ainsi étant donné un ensemble de clauses S, il est possible d'obtenir un couple $\langle T, U \rangle$ où U est l'ensemble des impliqués unitaires de S et T est l'ensemble de clauses obtenu en simplifiant S avec les équations de U. Le calcul $\mathcal{K}$ peut alors être utilisé pour engendrer T', l'ensemble des impliqués premiers de T. Il est ensuite nécessaire d'effectuer des opérations supplémentaires pour engendrer les impliqués premiers de S. En effet, cet ensemble est distinct de $T' \cup U$ puisque, par exemple, les clauses engendrées avec des prémisses respectivement dans T' et dans U ne sont pas nécessairement dans $T' \cup U$. Les impliqués premiers de S peuvent être récupérés en calculant la saturation de $T' \cup S$ par $\mathcal{K}$, mais nous avons défini un algorithme plus efficace permettant de récupérer ces impliqués. Le principe de l'algorithme repose sur le fait qu'il est inutile d'appliquer les règles d'inférence de $\mathcal{K}$ quand aucun des prémisses n'est dans U, ce qui réduit de façon drastique le nombre d'inférences à effectuer pour récupérer l'ensemble des impliqués premiers.

Afin d'améliorer l'efficacité de cette procédure, nous avons exploré dans [78] comment l'adapter afin de pouvoir traiter les impliqués unitaires à la volée, c'est-à-dire d'appliquer uniquement le calcul $\mathcal{K}$ et de simplifier l'ensemble de clauses en entrée au fur et à mesure que des impliqués unitaires sont engendrés. Maintenir la complétude du calcul en ajoutant cette possibilité de simplifier les clauses n'est pas trivial car il est parfois nécessaire d'effectuer des inférences avec pour prémisses des clauses simplifiées dont la version d'origine a déjà été traitée (la version d'origine est dans l'ensemble P de l'Algorithme 1). La complétude du calcul peut être rétablie en transférant toutes les clauses simplifiées de l'ensemble P à W, mais cette solution n'est pas satisfaisante. Nous avons donc défini la notion de $\langle a, b \rangle$-*neutralité* et prouvé qu'il est possible de maintenir la complétude du calcul en ne transférant de P à W que les clauses qui ne sont pas $\langle a, b \rangle$-neutres. Nous avons mené une étude expérimentale en comparant les performances de l'outil `Kparam` auquel a été ajoutée la gestion des impliqués unitaires avec `Zres` et avons constaté une nette amélioration du premier, qui devient au moins deux fois plus rapide que `Zres` dans 73% des cas, contre 54% auparavant.

6.2 Calcul de superposition contraint

6.2.1 Cas des abducibles constants

Nous avons choisi dans [77, 85] d'explorer une nouvelle approche à la génération d'impliqués premiers, basée sur un calcul de superposition contraint. Le principe de ce calcul est de permettre d'ajouter des hypothèses à des ensembles de clauses dans le processus de saturation du calcul de Superposition standard. Ces hypothèses supplémentaires sont conservées dans des contraintes associées aux clauses et propagées lors des inférences ; intuitivement, si une clause de la forme $[\![C \mid \mathcal{X}]\!]$ est engendrée à partir d'un ensemble S, ceci signifie que la clause standard C peut être engendrée à partir de $S \cup \mathcal{X}$. En particulier, si une clause de la forme $[\![\Box \mid \mathcal{X}]\!]$ est engendrée, alors $S \cup \mathcal{X}$ est insatisfaisable, ce qui signifie que $\mathcal{X}^c$ est un impliqué de S.

Le fait de stocker les hypothèses supplémentaires dans des contraintes et non pas directement dans les clauses, comme c'était le cas pour le calcul $\mathcal{K}$, tend à augmenter l'espace de recherche dans la génération des impliqués premiers. En effet, une clause contenant n littéraux peut être représentée de 2^n façons différentes, en fonction des littéraux qui sont stockés ou non dans les contraintes. Nous avons néanmoins choisi d'explorer cette approche car le fait de stocker les hypothèses dans les contraintes apporte deux avantages principaux :

- Cela permet d'utiliser les contraintes d'ordre et les fonctions de sélection standard de $\mathcal{SP}$ dans le calcul contraint tout en maintenant la complétude de ce dernier, ce qui n'était pas le cas avec le calcul $\mathcal{K}$.
- Cela permet d'imposer des restrictions sur les impliqués premiers à engendrer, ce qui peut être un avantage pratique quand l'ensemble des impliqués premiers d'un ensemble donné est extrêmement large.

D'un point de vue formel, nous considérons donc des clauses contraintes de la forme $[\![C \mid \mathcal{X}]\!]$, où C est une clause standard et$\mathcal{X}$ est un $\mathcal{A}$-*ensemble*, c'est-à-dire un ensemble de littéraux qui contiennent comme termes fermés des éléments de $\mathcal{A}$ uniquement. Ces clauses contraintes sont appelées des $\mathcal{A}$-*clauses*. Le calcul de superposition est adapté à ces $\mathcal{A}$-clauses de façon naturelle : à une inférence standard de la forme

$$\frac{C_1 \quad \cdots \quad C_n}{D\sigma}$$

est associée une inférence

$$\frac{[\![C_1 \mid \mathcal{X}_1]\!] \quad \cdots \quad [\![C_n \mid \mathcal{X}_n]\!]}{[\![D \mid \mathcal{X}_1 \cup \ldots \cup \mathcal{X}_n \cup \mathcal{Y}]\!]\,\mu},$$

avec la différence notable qu'au lieu d'appliquer un unificateur le plus général standard des termes $\langle t_1, \ldots, t_n \rangle$ à la conclusion de la règle d'inférence, nous déterminons une substitution la plus générale μ et un $\mathcal{A}$-ensemble $\mathcal{Y}$ tels que

Assertion positive $\dfrac{[\![t \bowtie s \vee C \mid \mathcal{X}]\!]}{[\![u \bowtie s \vee C \mid \mathcal{X} \wedge t \simeq u]\!]}$

où :

$\bowtie \in \{\simeq, \not\simeq\}$, $s \prec_{\mathcal{A}} t$, $u \prec_{\mathcal{A}} t$ et $t \bowtie s$ est sélectionné dans $t \bowtie s \vee C$.

Assertion négative $\dfrac{[\![t \simeq s \vee C \mid \mathcal{X}]\!]}{[\![s \not\simeq u \vee C \mid \mathcal{X} \wedge t \not\simeq u]\!]}$

où :

$s \prec_{\mathcal{A}} t$, $u \prec_{\mathcal{A}} t$ et $t \simeq s$ est sélectionné dans $t \simeq s \vee C$.

FIGURE 6.4 – Transfert de littéraux sélectionnés

les $t_i\mu$ sont égaux modulo les égalités qui apparaissent dans $\mathcal{Y}$; on dit alors que les t_i sont *$\mathcal{A}$-unifiables*.

Exemple 12 Soit $\mathcal{A} \stackrel{\text{def}}{=} \{a, b, c\}$, et considérons les termes $t_1 \stackrel{\text{def}}{=} f(a, b, d)$, $t_2 \stackrel{\text{def}}{=} f(x, y, z)$, $t_3 \stackrel{\text{def}}{=} f(x, x, z)$ et $t_4 \stackrel{\text{def}}{=} f(x, x, x)$. Les termes t_1 et t_2 sont unifiables (et donc $\mathcal{A}$-unifiables) ; t_1 et t_3 sont $\mathcal{A}$-unifiables ($\mu \stackrel{\text{def}}{=} \{x \leftarrow a, z \leftarrow d\}$ et $\mathcal{Y} \stackrel{\text{def}}{=} \{a \simeq b\}$) ; t_1 et t_4 ne sont pas $\mathcal{A}$-unifiables, car d n'est pas un symbole abducible. ♣

Nous obtenons le calcul c$\mathcal{SP}_0$ en ajoutant à l'adaptation des règles de $\mathcal{SP}$ des règles d'inférence permettant le transfert de littéraux sélectionnés vers les contraintes (voir la Figure 6.4). Après avoir défini des notions de redondance et de saturation sur les $\mathcal{A}$-clauses qui coïncident avec les notions usuelles sur les clauses standard, nous avons démontré que si S est un ensemble de $\mathcal{A}$-clauses saturé par c$\mathcal{SP}_0$ et C est une clause ne contenant que des symboles abducibles, telle que $S \models C$, alors S contient une $\mathcal{A}$-clause de la forme $[\![\Box \mid \mathcal{Y}]\!]$ telle que $\mathcal{Y}^{\text{c}} \models C$. Autrement dit, l'ensemble $\{\mathcal{Y}^{\text{c}} \mid [\![\Box \mid \mathcal{Y}]\!] \in S\}$ est l'ensemble des $\mathcal{A}$-impliqués premiers de S.

Le calcul c$\mathcal{SP}_0$ a été implémenté en OCaml, dans un outil appelé `csp`$_0$ [3]. Bien que le fait d'utiliser des clauses contraintes implique qu'il y a plus de clauses à considérer qu'avec le calcul $\mathcal{K}$, les expériences menées ont montré que l'utilisation de contraintes d'ordre et d'une fonction de sélection permet d'engendrer les $\mathcal{A}$-impliqués premiers à l'aide de c$\mathcal{SP}_0$ dans une majorité des cas ; de plus, sur les 1000 ensembles de clauses testés, `csp`$_0$ engendre les impliqués premiers dix fois plus rapidement que `Kparam` dans 25% des cas.

Le calcul c$\mathcal{SP}_0$ peut également être adapté à moindre coût pour n'engendrer que les impliqués premiers vérifiant certaines conditions. Chercher à imposer des conditions sur les impliqués premiers engendrés se justifie par le fait que la totalité des impliqués premiers d'un ensemble de clauses peut être très large, pouvant dépasser le million d'éléments alors que l'ensemble

3. http://lig-membres.imag.fr/tourret/index.php?&tab=3&slt=tools

de clauses de départ ne contient que peu de clauses [4]. Il est donc naturel , pour quelqu'un cherchant à engendrer les explications d'une formule satisfaisable, non seulement de spécifier la forme de ces explications (en fixant l'ensemble des symboles abducibles), mais également d'imposer des contraintes sur ces explications. Ces contraintes peuvent être syntaxiques, comme par exemple des bornes sur la taille des impliqués engendrés, ou bien sémantiques, comme par exemple d'imposer que chaque impliqué ait une clause de l'ensemble d'origine pour conséquence logique. Plutôt que de récupérer ces impliqués *a posteriori* en filtrant la totalité des impliqués, nous avons défini une condition de *fermeture par subsomption* sur les contraintes apparaissant dans les $\mathcal{A}$-clauses, et prouvé que l'adaptation de c$\mathcal{SP}_0$ telle que toutes les inférences engendrant une $\mathcal{A}$-clause dont la contrainte ne vérifie pas la condition souhaitée, permet d'engendrer les impliqués premiers vérifiant la condition en question, et eux seuls. Cette technique permet par exemple, d'engendrer uniquement les impliqués premiers d'une taille bornée, qui sont positifs, négatifs, ou encore des implicants d'une formule donnée, ou n'importe quelle combinaison de ces conditions.

6.2.2 Cas des abducibles quelconques

Nous avons défini dans [80, 89] un calcul, c$\mathcal{SP}$, permettant d'engendrer des $\mathcal{A}$-impliqués premiers dans le cas où $\mathcal{A}$ peut contenir des symboles autres que des constantes. Bien qu'étant basé sur des clauses contraintes, ce calcul ne peut pas être considéré comme une simple adaptation de c$\mathcal{SP}_0$. Les contraintes sont définies comme des ensembles quelconques de littéraux dont la sémantique est similaire à celle définie précédemment et le calcul est paramétré par un ensemble $\mathfrak{X}$ de contraintes fermées par inclusion, ce qui permet de contrôler les impliqués engendrés en bloquant les inférences qui génèrent des clauses dont les contraintes ne sont pas dans $\mathfrak{X}$. Il est ainsi possible d'engendrer les $\mathcal{A}$-impliqués d'un ensemble en considérant le cas où $\mathfrak{X}$ est l'ensemble des contraintes construites uniquement avec des symboles de $\mathcal{A}$, et d'autres restrictions comme par exemple sur la taille et la polarité des contraintes peuvent également être imposées. L'ensemble $\mathfrak{X}$ permet également d'engendrer des impliqués premiers quantifiés universellement ; il suffit pour cela que cet ensemble contienne la forme Skolemisée des contraintes correspondantes, c'est-à-dire des contraintes avec des constantes n'apparaissant pas dans l'ensemble de clauses considéré. Le calcul est également une adaptation de $\mathcal{SP}$, mais il est plus focalisé que c$\mathcal{SP}_0$. Ainsi, l'adaptation des règles de $\mathcal{SP}$ est plus directe : à une règle de $\mathcal{SP}$ de la forme

$$\frac{C_1 \quad \cdots \quad C_n}{D\sigma}$$

4. Il est fréquent pour des ensembles contenant moins d'une dizaine de clauses, chacune constituée de moins d'une dizaine de littéraux, d'admettre plusieurs millions d'impliqués premiers.

$$\textit{Assertion Positive} \quad \frac{[\![u[t] \bowtie v \vee C \mid \mathcal{X}]\!]}{[\![u[s] \bowtie v \vee C \mid \mathcal{X} \wedge t' \simeq s]\!]\,\sigma} \qquad (i), (iv)$$

$$\textit{Assertion Négative} \quad \frac{[\![t \simeq s \vee C \mid \mathcal{X}]\!]}{[\![u[s] \bowtie v \vee C \mid \mathcal{X} \wedge u[t'] \bowtie v]\!]\,\sigma} \qquad (ii), (iii), (v)$$

où $\bowtie$ représente $\simeq$ ou bien $\not\simeq$, et les conditions suivantes sont vérifiées :

Pour toutes les règles, $\sigma = \mathrm{mgu}(t, t')$;

(i) : $s \prec t'$, $v\sigma \not\succ u[t]\sigma$ et $(u[t] \bowtie v)\sigma \in \mathrm{sel}((u[t] \bowtie v \vee C)\sigma)$;

(ii) : $(t \simeq s)\sigma \in \mathrm{sel}((t \simeq s \vee C)\sigma)$ et $s\sigma \not\succ t\sigma$;

(iii) : $v \prec u[t']$;

(iv) : $\mathcal{X} \wedge t' \simeq s \in \mathfrak{X}$;

(v) : $\mathcal{X} \wedge u[t'] \bowtie v \in \mathfrak{X}$.

FIGURE 6.5 – Règles d'inférence pour c$\mathcal{SP}$.

est associée une règle de c$\mathcal{SP}$ de la forme

$$\frac{[\![C_1 \mid \mathcal{X}_1]\!] \quad \cdots \quad [\![C_n \mid \mathcal{X}_n]\!]}{[\![D \mid \mathcal{X}_1 \cup \ldots \cup \mathcal{X}_n]\!]\,\sigma}$$

Ce calcul autorise donc moins d'inférences car les termes considérés dans les prémisses des règles doivent être unifiables et non plus $\mathcal{A}$-unifiables. c$\mathcal{SP}$ permet d'employer le même ordre de simplification et la même fonction de sélection que $\mathcal{SP}$, et l'ajout d'hypothèses à des contraintes est plus dirigé que dans le cas de c$\mathcal{SP}_0$: une hypothèse est ajoutée à une clause contrainte seulement s'il est possible d'effectuer une inférence dans $\mathcal{SP}$ avec pour prémisses la clause en question. Les règles d'ajout d'hypothèses aux contraintes sont résumées dans la Figure 6.5.

La notion de redondance employée dans c$\mathcal{SP}$ est une adaptation directe de celle dans $\mathcal{SP}$, tenant compte des contraintes, et nous avons réduit l'espace de recherche en normalisant systématiquement les clauses contraintes engendrées par le calcul. Nous avons démontré que c$\mathcal{SP}$ est correct et complet : si S est un ensemble de clauses et S' est une saturation de S par c$\mathcal{SP}$, alors pour toute contrainte $\mathcal{X} \in \mathfrak{X}$, $S \models \mathcal{X}^{\mathrm{c}}$ si et seulement s'il existe une clause contrainte de la forme $[\![\Box \mid \mathcal{X}']\!]$ dans S' telle que $\mathcal{X}' \subseteq \mathcal{X}$.

Nous avons réalisé une implémentation de ce calcul, nommée `csp`[5], dans le cas où les ensembles de clauses considérés sont fermés. Dans cette implémentation, nous avons employé un critère de redondance plus restreint que celui utilisé dans la preuve de la complétude de c$\mathcal{SP}$, et avons adapté les

5. http://lig-membres.imag.fr/tourret/index.php?&tab=3&slt=tools

arbres clausaux afin de pouvoir tenir compte des contraintes associées aux clauses ; obtenant ainsi les arbres clausaux contraints. Nous avons défini les algorithmes permettant de tester si un arbre clausal contraint contient une clause contrainte plus générale qu'une clause contrainte donnée, et, dans le cas contraire, d'insérer cette clause dans l'arbre tout en en supprimant les clauses moins générales que celle nouvellement insérée. L'outil résultant a été comparé à $\texttt{csp}_0$, `Zres` et `primer`, un outil [6] plus récent permettant d'engendrer les impliqués propositionnels plus efficacement que `Zres` (voir par exemple [148]). Pour effectuer les comparaisons entre les différents outils, les problèmes considérés ont été réduits à la logique équationnelle avec uniquement des constantes et à la logique propositionnelle. Nous avons comparé les temps nécessaires aux différents outils pour engendrer les impliqués premiers, sans tenir compte des post-traitements nécessaires à la reconstitution des impliqués premiers de la formule d'origine. Quand les problèmes initiaux ne contiennent que des constantes, $\texttt{csp}_0$ tend à être l'outil le plus performant, ce qui s'explique par le fait que les algorithmes de stockage dans des arbres clausaux sont alors plus simples et plus efficaces. Dans le cas général cependant, `csp` est clairement l'outil le plus efficace pour engendrer les impliqués premiers.

6.3 Discussion

Nos travaux ont mené à la définition de différents calculs permettant d'engendrer des impliqués premiers par saturation, et, par dualité, de construire des explications de façon automatique. Ces calculs sont complets ; il est garanti que chacun d'entre eux permet d'engendrer les impliqués premiers de la forme appropriée (construits à partir de constantes, de termes fermés ou de termes quelconques), quel que soit l'ensemble de clauses considéré, qu'il soit fermé ou non. L'opération de normalisation de clauses, que nous avons défini lors de notre étude de critères syntaxiques pour détecter efficacement les clauses redondantes, pourrait potentiellement être utilisée dans des démonstrateurs standard basés sur la superpositions. Des études expérimentales ont en effet montré que dans certains cas, la normalisation systématique des clauses considérées lors du processus de saturation permet de réduire le nombre de clauses à vérifier d'un ordre de magnitude. Il serait intéressant de pousser ces expérimentations pour étudier précisément l'impact qu'aurait cette normalisation systématique sur les performances des démonstrateurs automatiques, après l'avoir étendue aux clauses quelconques et non plus seulement aux clauses fermées. Le fait de pouvoir imposer des contraintes que doivent vérifier les impliqués premiers obtenus et de garantir que les calculs engendrent exactement ces derniers sans nécessiter de post-traitement est d'un grand intérêt, puisque le nombre d'impliqués premiers d'un en-

6. http://logos.ucd.ie/web/doku.php?id=primer

semble de clauses peut être très élevé, et qu'il peut sembler raisonnable de, par exemple, limiter la taille et/ou la polarité des impliqués recherchés.

Chapitre 7

Perspectives

Les travaux menés dans plusieurs des thématiques abordées dans ce mémoire ne sont pas finalisés et continuent à donner lieu à des pistes que nous explorons encore.

7.1 Intégration du raisonnement inductif dans des procédures basées sur la saturation

Nous étudions en ce moment l'intégration du raisonnement inductif dans les procédures de preuve basées sur la saturation. Le but de ces travaux est d'être capable de démontrer des théorèmes nécessitant l'emploi de l'induction en se servant de systèmes d'inférence complets pour la réfutation. Le principe de la méthode sur laquelle nous travaillons est d'étendre ces systèmes d'inférence pour qu'ils puissent *automatiquement* engendrer des conséquences inductives de la formule fournie en entrée, puis d'en construire une réfutation avec les règles d'inférence standard. Par exemple, sur l'ensemble $S = \{p(a), \neg p(x) \vee p(s(x)), \neg p(n)\}$, où n doit être interprété comme un terme de la forme $s^k(a)$ avec $k \in \mathbb{N}$, le calcul engendre la formule $\forall x\, p(x)$. Cette dernière est une conséquence inductive de S, et l'ajout de la clause $p(x)$ à S permet d'engendrer la clause vide à l'aide de règles d'inférence standard.

Dans le cas général, la génération de la conséquence inductive permettant de construire une réfutation s'effectue en plusieurs étapes que nous décrivons ci-dessous. Dans un souci de simplification, nous supposons que l'ensemble contient un unique terme t qui doit être interprété sur un domaine inductif dont les constructeurs sont a et f.

- Le terme t est abstrait des clauses où il apparaît, ce qui permet d'appliquer les règles d'inférence standard sur des clauses contraintes, les contraintes conservant les propriétés que doivent vérifier ces termes inductifs. Par exemple, la clause $p(b) \vee q(t)$ est remplacée par la clause contrainte $[\![p(b) \vee q(x) \mid x \simeq t]\!]$.
- Une réfutation de la forme $[\![\Box \mid t \simeq a]\!]$ est engendrée, ce qui signifie

que l'ensemble S est effectivement insatisfaisable si $t = a$. La procédure extrait alors de cette réfutation une formule $\varphi(x)$ telle que $\neg\varphi(a)$ a été employée pour engendrer la réfutation. La formule $\varphi(x)$ est potentiellement un invariant inductif.
- La procédure permet de vérifier que $\varphi(x)$ est effectivement un invariant inductif en appliquant le principe de descente infinie ; autrement dit, en prouvant que si $\varphi(t)$ n'est pas vraie, alors il existe s un sous-terme strict de t tel que $\varphi(s)$ n'est pas vraie. Si c'est le cas, alors l'invariant $\forall x\varphi(x)$ est ajouté à l'ensemble de clauses, et si le principe de la descente infinie échoue, alors le calcul extrait de cet échec l'information permettant de raffiner l'invariant candidat avant d'appliquer à nouveau le principe de la descente infinie.

Une fois que les règles permettant d'engendrer les invariants inductifs auront été précisément définies et que leur correction aura été démontrée, ces règles pourront être intégrées à des systèmes d'inférence comme le calcul de Superposition afin de réaliser des études expérimentales sur leur efficacité. Il sera également intéressant d'identifier des classes de formules pour lesquelles il est garanti que le calcul est complet, et plus particulièrement de critères syntaxiques assurant la complétude du calcul.

7.2 Méthodes de décomposition pour l'abduction

Une des limitations de nos travaux sur les procédures d'abduction est que, à l'instar des outils basés sur la superposition en démonstration automatique, ceux que nous avons développés ne sont pas capables de traiter efficacement les formules avec une structure combinatoire trop importante. C'est pourquoi nous avons lancé un travail de recherche sur la génération d'impliqués premiers par des méthodes basées sur la décomposition, dans le but d'adapter les techniques mises au point pour la résolution efficace de problèmes SAT (et implémentés dans les solveurs CDCL), au cadre de la génération d'impliqués premiers. L'objectif est de concevoir des outils capables de gérer des structures combinatoires importantes, tout en laissant aux utilisateurs la possibilité de choisir les ensembles de symboles abducibles et les contraintes que doivent satisfaire les impliqués premiers engendrés. Nous avons débuté ces travaux dans le cadre propositionnel, et c'est sur ce thème qu'a travaillé Yanis Sellami lors de son stage de M2R (voir [158]). Nous avons conçu un outil basé sur le solveur SAT Mini-SAT [90] qui permet une énumération et une sélection efficace des impliqués d'un ensemble de clauses propositionnelles. L'algorithme mis au point est sommairement décrit dans l'Algorithme 2. Cet algorithme récursif prend initialement en entrée l'ensemble des clauses dont il faut déterminer les impliqués premiers, une conjonction vide d'abducibles et un ensemble d'abducibles candidats. A chaque appel récursif, un nouvel abducible candidat est ajouté à la conjonction d'abducibles. Si $S \cup M$ est

Algorithme 2 : PIdec : Enumération d'impliqués premiers

entrée : un ensemble de clauses S
entrée : une conjonction d'abducibles M
entrée : un ensemble d'abducibles candidats A
sortie : les impliqués premiers de S
si $S \cup M \models \square$ **alors**
  $M' := \texttt{Minimiser}(M, S)$;
  retourner $\{\neg M'\}$
sinon
  Soit I un modèle de $S \cup M$;
  $B := \texttt{Filtrer}(A, I)$;
  pour chaque $l \in B$ **faire**
    $P_l := \text{PIdec}(S,\ M \wedge l,\ B \setminus \{l\})$
  retourner $\bigcup_{l \in B} P_l$

insatisfaisable alors $\neg M$ est un impliqué de S, et un impliqué premier de S est extrait de M grâce à la fonction `Minimiser`. Sinon, un modèle de $S \cup M$ est utilisé par la fonction `Filtrer` pour réduire le plus possible les abducibles candidats à considérer dans les appels récursifs.

Des premières études expérimentales ont montré que, si cet outil est plus efficace que `Zres`, l'outil `primer` reste le plus performant pour engendrer tous les impliqués d'un ensemble de clauses donné. L'outil que nous avons conçu possède néanmoins plus de fonctionnalités que `primer`, puisqu'il permet une sélection des symboles abducibles sur lesquels doivent être construits les impliqués, ainsi que l'imposition de contraintes pour filtrer ces impliqués.

L'étape suivante est d'adapter ces travaux à des logiques plus expressives, en remplaçant le solveur SAT par un solveur SMT. La structure combinatoire des problèmes sera la même que dans le cas propositionnel, mais la minimisation des impliqués engendrés soulève de nombreux problèmes théoriques : si C et D sont des impliqués tels que $C \models D$, alors en logique propositionnelle, tous les littéraux apparaissant dans C apparaissent également dans D. Ceci n'est plus le cas dans des logiques plus expressives, où les impliqués pourront de surcroît être construits modulo des théories, et l'étape de minimisation des impliqués sera donc plus complexe à mettre en oeuvre. Nous allons débuter ces travaux d'extension en logique équationnelle du premier ordre.

Nos travaux sur l'abduction et la génération automatique d'hypothèses ont d'autres applications que la détection d'erreurs, et nous travaillons sur une de ces applications dans le cadre de la logique de séparation. Cette logique a été développée pour pouvoir raisonner sur des programmes manipulant la mémoire, et contenant donc des structures dont les champs peuvent être modifiés [153]. Certifier que de tels programmes sont corrects nécessite

souvent de prouver des conjectures en logique de séparation nécessitant des invariants inductifs, et nous étudions comment utiliser le raisonnement abductif pour identifier automatiquement les hypothèses qui permettraient de prouver de telles conjectures.

Nous travaillons sur ces thèmes dans le cadre de la thèse de Yanis Sellami, que nous co-encadrons avec Nicolas Peltier, et nous collaborons en ce moment avec Radu Iosif (Laboratoire Verimag), ainsi qu'avec Stéphane Demri et Etienne Lozes (Laboratoire Spécification et Vérification de l'ENS Cachan) sur la logique de séparation.

7.3 Assistants de preuve

Les travaux sur une automatisation plus importante du raisonnement sont complémentaires à ceux sur l'aide au raisonnement, dont le but est de concevoir des assistants de preuve capables de vérifier et guider les preuves effectuées par des humains. Dans ce domaine, les nombreux travaux de recherche sur les assistants de preuve ont permis la formalisation de nombreux théorèmes mathématiques, dont l'un des plus célèbres est le théorème des quatre couleurs, dont la preuve en Coq a été construite par Georges Gonthier et Benjamin Werner [99]. De nombreuses améliorations, et notamment l'intégration d'outils externes ont permis de déléguer des parties de plus en plus importantes des preuves aux machines. Ainsi, des outils comme Sledgehammer [143] ou Nitpick [30] qui sont intégrés à l'assistant de preuve Isabelle [142] et peuvent être utilisés pour respectivement construire des preuves et des contre-exemples permettent aux utilisateurs de formaliser de preuves de plus en plus efficacement. Nous estimons que les assistants de preuve devraient dans un futur proche être utilisés de façon plus conséquente par les mathématiciens pour la réalisation de preuves de plus en plus complexes. Pour que ceci soit possible, nous pensons que les assistants de preuve doivent évoluer dans deux directions principales.

- Il faut créer pour ces outils des interfaces qui soient les plus intuitives possible. Une raison qui pourrait expliquer qu'il n'y a pas plus de mathématiciens se servant d'assistants de preuve est que les premiers peuvent considérer le coût d'entrée comme trop élevé. Il semble donc nécessaire de poursuivre l'effort visant à rendre ces outils accessibles à des non-initiés, en poursuivant par exemple le développement de langages de preuves faciles à lire comme Isar [176, 69], ou d'environnements de développement intuitifs comme jEdit [175].
- Il faut poursuivre l'effort de coopération entre des assistants de preuve et des outils entièrement automatisés, capables d'effectuer des preuves en logique du premier ordre, par induction, ou encore de construire des contre-exemples à des formules ou d'expliquer pourquoi une formule n'est pas valide. Idéalement, cette coopération devrait nécessiter

une intégration la plus légère possible, pour que différents assistants puissent être connectés à différents outils avec un effort minimal.

Nous avons pour but à plus long terme d'adapter les outils que nous avons conçus pour le raisonnement par induction et abduction, afin de les intégrer dans des assistants de preuve comme Isabelle [142] ou bien Coq [165]. Nos outils pour le raisonnement inductif, et notamment celui basé sur les procédés de saturation, pourraient être intégrés assez facilement dans Isabelle grâce à Sledgehammer [29], qui est déjà employé pour interagir avec des démonstrateurs automatiques dont CVC4 [1], E [2], Spass [3], Z3 [4] ou Vampire [5]. Pour qu'ils soient d'un intérêt plus pratique, nous avons l'intention d'étendre nos résultats sur le raisonnement abductif à la logique d'ordre supérieur. Notre objectif sera d'obtenir des procédures d'abduction efficaces, même si elles ne peuvent pas être complètes, afin de guider l'utilisateur dans la recherche d'hypothèses manquantes pour démontrer des théorèmes.

Nous pourrons expérimenter ces outils tout au long du processus de formalisation des mathématiques financières en Isabelle que nous avons démarré récemment, en collaboration avec Nicolas Peltier, Hervé Guiol (Laboratoire TIMC) et Jérôme Lelong (Laboratoire LJK). Les mathématiques financières ont pris leur essor après la démonstration en 1973 par Fischer Black, Myron Scholes et Robert Merton que, sous certaines hypothèses sur l'évolution aléatoire d'un actif financier, il est possible de trouver le juste prix de produits dérivés dont le sous-jacent est l'actif financier [27, 129]. Ce prix est déterminé en montrant qu'il est possible de définir une stratégie d'investissement (appelée portefeuille de couverture) qui permet de répliquer le produit dérivé, c'est-à-dire de générer exactement la richesse attendue à la date donnée. Cela fournit ainsi une technique permettant de se couvrir contre les risques liés à l'évolution du marché. Depuis, les travaux de plusieurs chercheurs ont permis la définition de nombreux modèles pouvant être utilisés pour trouver le juste prix de produits dérivés sur les différents marchés : action, taux de change, taux d'intérêt, crédit ou matières premières. Ces modèles sont implémentés dans des programmes appelés pricers, et utilisés par différentes institutions financières. Ces pricers représentent un risque opérationnel important, car la moindre erreur d'implémentation peut causer de grandes pertes financières. C'est pourquoi les institutions mettent en place des procédures de validation dans les départements de gestion des risques, pour s'assurer que les modèles sont correctement implémentés dans les pricers. Ceci reste une tâche difficile, et il n'est pas aisé de détecter les erreurs numériques, notamment en raison des aléas et incertitudes liés aux simulations. L'objectif de nos travaux sur ce thème est de concevoir à l'aide d'Isabelle des pricers certifiés, c'est-à-dire

1. http://cvc4.cs.stanford.edu/web/
2. http://eprover.org
3. http://spass-prover.org
4. https://github.com/Z3Prover/z3
5. http://www.vprover.org/

des pricers pour lesquels il est démontré formellement qu'ils ne présentent pas d'erreurs.

Nous avons débuté ces travaux dans [88], en nous plaçant dans le cadre d'un modèle en temps discret, plus précisément le modèle binomial dû à Cox, Ross et Rubinstein [57]. Ce modèle est de loin le plus simple d'un point de vue mathématique et présente également l'avantage de permettre une résolution explicite. Il fournit une méthode permettant d'évaluer le prix des options, modulo certaines hypothèses sur l'évolution du marché. Les résultats obtenus lorsque l'on applique cette méthode à des options européennes (c'est-à-dire des options exerçables uniquement à une date précise) convergent vers ceux donnés par le modèle continu dû à Black et Scholes, qui est beaucoup plus complexe. Cette méthode est aussi fréquemment employée pour valoriser des produits pouvant être exercés avant échéance (options dites américaines). Une première formalisation de ce modèle dans l'assistant de preuve Isabelle a déjà été réalisée[6]. Bien que ce modèle mathématique soit relativement simple, sa formalisation en Isabelle a nécessité plusieurs milliers de lignes.

Nous avons l'intention dans un premier temps d'étendre les premiers résultats obtenus, toujours dans le cas discret. Une première application serait l'obtention d'un pricer certifié pour options américaines dans le modèle de Cox, Ross et Rubinstein. La formalisation serait réalisée d'une façon aussi générique et abstraite que possible, de telle sorte qu'une partie des résultats obtenus, par exemple la formalisation de notions telles que l'absence d'arbitrage (qui garantit l'absence d'opportunité de profit sans risque sur un marché) pourrait ensuite être ré-utilisée dans le cadre de modèles plus complexes. Dans un second temps, nous prévoyons de commencer la formalisation des notions et résultats fondamentaux du calcul stochastique, qui servent de base aux traitements des modèles en temps continu, en particulier le mouvement brownien et l'intégrale stochastique d'Itô, processus Gaussien, changement de probabilités, martingales et leur représentation.

6. http://crr-isabelle.forge.imag.fr/

Bibliographie

[1] Aharon Abadi, Alexander Rabinovich, and Mooly Sagiv. Decidable fragments of many-sorted logic. *J. Symb. Comput.*, 45(2) :153–172, 2010. 34

[2] Wilhelm Ackermann. Über die Erfüllbarkeit gewisser Zählausdrücke. *Mathematische Annalen*, 100 :638–649, 1928. 18

[3] Markus Aderhold. Improvements in formula generalization. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pages 231–246, 2007. 49

[4] Ernst Althaus, Evgeny Kruglov, and Christoph Weidenbach. Superposition modulo linear arithmetic SUP(LA). In S. Ghilardi and R. Sebastiani, editors, *FroCoS 2009*, volume 5749 of *LNCS*, pages 84–99. Springer, 2009. 29, 39

[5] V. Aravantinos, R. Caferra, and N. Peltier. A schemata calculus for propositional logic. In *TABLEAUX 09 (International Conference on Automated Reasoning with Analytic Tableaux and Related Methods)*, volume 5607 of *LNCS*, pages 32–46. Springer, 2009. 51

[6] V. Aravantinos, R. Caferra, and N. Peltier. A Decidable Class of Nested Iterated Schemata. In *IJCAR 2010 (International Joint Conference on Automated Reasoning)*, LNCS, pages 293–308. Springer, 2010. 51

[7] V. Aravantinos, R. Caferra, and N. Peltier. RegSTAB : a SAT-Solver for Propositional Schemata. In *IJCAR 2010 (International Joint Conference on Automated Reasoning)*, LNCS, pages 309–315. Springer, 2010. 51

[8] V. Aravantinos, R. Caferra, and N. Peltier. Decidability and undecidability results for propositional schemata. *Journal of Artificial Intelligence Research*, 40 :599–656, 2011. 51

[9] Vincent Aravantinos, Mnacho Echenim, and Nicolas Peltier. A resolution calculus for first-order schemata. *Fundamenta Informaticae*, 125(2) :101–133, 2013. 12, 53, 54

[10] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic*, 10(1) :129–179, January 2009. 22, 23, 25, 28, 29, 37, 63

[11] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2) :140–164, 2003. 22, 23, 25, 28, 29, 37, 63

[12] Jürgen Avenhaus, Ulrich Kühler, Tobias Schmidt-Samoa, and Claus-Peter Wirth. How to prove inductive theorems ? quodlibet ! In *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, pages 328–333, 2003. 50

[13] Franz Baader, Silvio Ghilardi, and Cesare Tinelli. A new combination procedure for the word problem that generalizes fusion decidability results in modal logics. *Inf. Comput.*, 204(10) :1413–1452, 2006. 21

[14] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. CERES : An analysis of Fürstenberg's proof of the infinity of primes. *Theor. Comput. Sci.*, 403(2-3) :160–175, 2008. 51

[15] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 3(4) :217–247, 1994. 10

[16] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Refutational theorem proving for hierachic first-order theories. *Appl. Algebra Eng. Commun. Comput.*, 5 :193–212, 1994. 39, 43, 62

[17] Clark Barrett, Leonardo Mendonça de Moura, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The SMT-LIB initiative and the rise of SMT - (HVC 2010 award talk). In Sharon Barner, Ian G. Harris, Daniel Kroening, and Orna Raz, editors, *Hardware and Software : Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*, volume 6504 of *Lecture Notes in Computer Science*, page 3. Springer, 2010. 20

[18] Clark Barrett, Morgan Deters, Leonardo Mendonça de Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *J. Autom. Reasoning*, 50(3) :243–277, 2013. 20

[19] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT.

In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer, 2002. 18

[20] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009. 17

[21] Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. In Franz Baader, editor, *CADE-19 – The 19th International Conference on Automated Deduction*, volume 2741 of *LNAI*, pages 350–364. Springer, 2003. 34

[22] Peter Baumgartner and Uwe Waldmann. Hierarchic superposition with weak abstraction. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898, pages 39–57. Springer, 2013. 29

[23] Paul Bernays and Moses Schönfinkel. Zum Entscheidungsproblem der Mathematischen Logik. *Mathematische Annalen*, 99 :342–372, 1928. 18

[24] Narjes Berregeb, Adel Bouhoula, and Michaël Rusinowitch. SPIKE-AC : A system for proofs by induction in associative-commutative theories. In Harald Ganzinger, editor, *Rewriting Techniques and Applications, 7th International Conference, RTA-96, New Brunswick, NJ, USA, July 27-30, 1996, Proceedings*, volume 1103 of *Lecture Notes in Computer Science*, pages 428–431. Springer, 1996. 49

[25] M. Bienvenu. Prime implicates and prime implicants in modal logic. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 379. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999, 2007. 60

[26] Susanne Biundo, B. Hummel, Dieter Hutter, and Christoph Walther. The karlsruhe induction theorem proving system. In Jörg H. Siekmann, editor, *8th International Conference on Automated Deduction, Oxford, England, July 27 - August 1, 1986, Proceedings*, Lecture Notes in Computer Science, pages 672–674. Springer, 1986. 48

[27] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3) :637–654, 1973. 79

[28] Patrick Blackburn, Johan Van Benthem, and Frank Wolter. *Handbook of Modal Logic*. Studies in logic and practical reasoning - ISSN 1570-2464 ; 3. Elsevier, 2007. 60

[29] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011. 79

[30] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick : A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 131–146, 2010. 78

[31] Maria Paola Bonacina and Mnacho Echenim. Rewrite-based decision procedures. *Electr. Notes Theor. Comput. Sci.*, 174(11) :27–45, 2007. 11, 26, 27, 29

[32] Maria Paola Bonacina and Mnacho Echenim. Rewrite-based satisfiability procedures for recursive data structures. *Electr. Notes Theor. Comput. Sci.*, 174(8) :55–70, 2007. 11, 23, 29

[33] Maria Paola Bonacina and Mnacho Echenim. T-decision by decomposition. In Frank Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 199–214. Springer, 2007. 11, 29

[34] Maria Paola Bonacina and Mnacho Echenim. On variable-inactivity and polynomial T-satisfiability procedures. *Journal of Logic and Computation*, 18(1) :77–96, 2008. 11, 24, 29, 63

[35] Maria Paola Bonacina and Mnacho Echenim. Theory decision by decomposition. *J. Symb. Comput.*, 45(2) :229–260, 2010. 11, 29, 33, 37

[36] Maria Paola Bonacina, Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Decidability and Undecidability results for Nelson-Oppen and rewrite-based decision procedures. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *LNCS*, pages 513–527. Springer, 2006. 23

[37] Adel Bouhoula, Emmanuel Kounalis, and Michaël Rusinowitch. SPIKE, an automatic theorem prover. In *Proceedings of LPAR'92*, volume 624, pages 460–462. Springer-Verlag, 1992. 48

[38] Adel Bouhoula, Emmanuel Kounalis, and Michaël Rusinowitch. Automated mathematical induction. *J. Log. Comput.*, 5(5) :631–668, 1995. 49

[39] Adel Bouhoula and Michaël Rusinowitch. Implicit induction in conditional theories. *J. Autom. Reasoning*, 14(2) :189–235, 1995. 49

[40] Thierry Boy de la tour, Mnacho Echenim, and Paliath Narendran. Unification and matching modulo leaf-permutative equational presentations. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *4th International Joint Conference, IJCAR 2008*, LNAI 5195, pages 332–347, Sydney, Australia, August 2008. Springer Verlag.

[41] Thierry Boy de la Tour, Mnacho Echenim, and Nicolas Peltier. Boolean abductive reasoning with symmetries. 2014.

[42] R. S. Boyer and J. S. Moore. *A computational logic*. Academic Press, 1979. 10, 48

[43] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation : Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. 34, 40

[44] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays ? In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proc. VMCAI-7*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006. 22, 33

[45] James Brotherston. Cyclic Proofs for First-Order Logic with Inductive Definitions. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods : Proceedings of TABLEAUX 2005*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005. 49

[46] James Brotherston. *Sequent calculus proof systems for inductive definitions*. PhD thesis, University of Edinburgh, 2006. 50

[47] Alan Bundy. Chapter 13 - the automation of proof by mathematical induction. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 845 – 911. North-Holland, Amsterdam, 2001. 48

[48] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling : meta-level guidance for mathematical reasoning*. Cambridge University Press, New York, NY, USA, 2005. 49

[49] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The Oyster-Clam system. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648, London, UK, 1990. Springer-Verlag. 48

[50] R. Caferra, A. Leitsch, and N. Peltier. *Automated Model Building*, volume 31 of *Applied Logic Series*. Kluwer Academic Publishers, 2004. 10, 58

[51] H. Chen, J. Hsiang, and H.C. Kong. On finite representations of infinite sequences of terms. In *Conditional and Typed Rewriting Systems, 2nd International Workshop*, pages 100–114. Springer, LNCS 516, 1990. 50

[52] Hong Chen and Jieh Hsiang. Recurrence domains : Their unification and application to logic programming. *Inf. Comput.*, 122(1) :45–69, 1995. 50

[53] Paula Chocron, Pascal Fontaine, and Christophe Ringeissen. A gentle non-disjoint combination of satisfiability procedures. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, pages 122–136, 2014. 21

[54] H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 14, pages 913–962. North-Holland, 2001. 49

[55] Hubert Comon and Robert Nieuwenhuis. Induction=i-axiomatization+first-order consistency. *Inf. Comput.*, 159(1-2) :151–186, 2000. 49

[56] O. Coudert and JC Madre. A new method to compute prime and essential prime implicants of boolean functions. In *Proc. of MIT VLSI Conference*, 1992. 59

[57] John C. Cox, Stephen A. Ross, and Mark Rubinstein. Option pricing : A simplified approach. *Journal of Financial Economics*, 7(3) :229–263, 1979. 80

[58] P T Cox and T Pietrzykowski. Causes for events : Their computation and applications. In *Proc. Of the 8th International Conference on Automated Deduction*, pages 608–621, New York, NY, USA, 1986. Springer-Verlag New York, Inc. 57

[59] Simon Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond*. PhD thesis, 2015. Thï¿½se de doctorat dirigï¿½e par Dowek, Gilles Informatique Palaiseau, Ecole polytechnique 2015. 29

[60] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962. 9, 19

[61] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3) :201–215, July 1960. 9, 10, 19

[62] Johan de Kleer. An improved incremental algorithm for generating prime implicates. In William R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992*, pages 780–785. AAAI Press / The MIT Press, 1992. 60

[63] Thierry Boy de la Tour and Mnacho Echenim. Solving linear constraints in elementary abelian p-groups of symmetries. *CoRR*, abs/1107.4553, 2011.

[64] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction : Automated Deduction*, CADE-21, pages 183–198, 2007. 22

[65] Leonardo de Moura and Nikolaj Bjørner. Engineering DPLL(T) + Saturation. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pages 475–490, 2008. 29

[66] David L. Detlefs, Greg Nelson, and James B. Saxe. Simplify : a theorem prover for program checking. *Journal of the ACM*, 52(3) :365–473, 2005. 22

[67] Peter J. Downey and Ravi Sethi. Assignment commands with array references. *J. ACM*, 25(4) :652–666, October 1978. 23

[68] Cvetan Dunchev, Alexander Leitsch, Mikheil Rukhaia, and Daniel Weller. CERES for first-order schemata. *CoRR*, abs/1303.4257, 2013. 51

[69] Sana Stojanovic Durdevic, Julien Narboux, and Predrag Janicic. Automated generation of machine verifiable and readable proofs : A case study of tarski's geometry. *Ann. Math. Artif. Intell.*, 74(3-4) :249–269, 2015. 78

[70] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for dpll(t). In Thomas Ball and Robert Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification*, pages 81–94, 2006. 22

[71] M. Echenim and N. Peltier. Instantiation of SMT problems modulo Integers. In *AISC 2010 (10th International Conference on Artificial Intelligence and Symbolic Computation)*, volume 6167 of *LNCS*, pages 49–63. Springer, 2010. 11, 38, 39, 41, 42

[72] M. Echenim and N. Peltier. A Calculus for Generating Ground Explanations. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'12)*, volume 7364, pages 194–209. Springer LNCS, 2012. 12, 61

[73] M. Echenim and N. Peltier. Reasoning on Schemata of Formulae. In *Proceedings of CICM 2012 (Conferences on Intelligent Computer Mathematics)*, volume 7362, pages 310–325. Springer LNCS, 2012. 12, 51

[74] M. Echenim and N. Peltier. Reasoning on Schemata of Formulae. Technical report, CoRR, abs/1204.2990, 2012. 51

[75] M. Echenim, N. Peltier, and S. Tourret. A Superposition Strategy for Abductive Reasoning in Ground Equational Logic. In *Proceedings of IWS 2012 (International Workshop on Strategies)*, 2012. 12, 63

[76] M. Echenim, N. Peltier, and S. Tourret. An approach to abductive reasoning in equational logic. In *Proceedings of IJCAI'13 (International Conference on Artificial Intelligence)*, pages 3–9. AAAI, 2013. 12, 63

[77] M. Echenim, N. Peltier, and S. Tourret. A Deductive-Complete Constrained Superposition Calculus for Ground Flat Equational Clauses. In *4th Workshop on Practical Aspects of Automated Reasoning*, 2014. 12, 68

[78] M. Echenim, N. Peltier, and S. Tourret. A Rewriting Strategy to Generate Prime Implicates in Equational Logic. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'14)*. Springer, 2014. 12, 67

[79] M. Echenim, N. Peltier, and S. Tourret. A Superposition-Based Approach to Abductive Reasoning in Equational Clausal Logic. In *Proceedings of ADDTC 2014 (Automated Deduction : Decidability, Complexity, Tractability)*, 2014. Invited talk (N. Peltier). 12

[80] M. Echenim, N. Peltier, and S. Tourret. Quantifier-Free Equational Logic and Prime Implicate Generation. In *CADE 25 (25nd International Conference on Automated Deduction)*. Springer, 2015. 12, 70

[81] Mnacho Echenim and Nicolas Peltier. Modular instantiation schemes. *Information Processing Letters*, 111(20) :989–993, 2011. 11, 41

[82] Mnacho Echenim and Nicolas Peltier. An instantiation scheme for satisfiability modulo theories. *Journal of Automated Reasoning*, 48(3), 2012. 11, 35, 38, 40, 41, 42

[83] Mnacho Echenim and Nicolas Peltier. Instantiation Schemes for Nested Theories. *ACM Transactions on Computational Logic*, 14(2) :11, 2013. 11, 42, 43

[84] Mnacho Echenim and Nicolas Peltier. On the construction of equational binary decision diagrams for equational formulæ. 2015.

[85] Mnacho Echenim and Nicolas Peltier. A Superposition Calculus for Abductive Reasoning. *Journal of Automated Reasoning*, 2016. 12, 68

[86] Mnacho Echenim and Nicolas Peltier. Combining induction and saturation-based theorem proving. 2016.

[87] Mnacho Echenim and Nicolas Peltier. Generating prime implicates by decomposition. 2016.

[88] Mnacho Echenim and Nicolas Peltier. The binomial pricing model in finance : A formalization in isabelle. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 546–562. Springer, 2017. 80

[89] Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. Prime Implicate Generation in Equational Logic. 2016. 70

[90] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. 76

[91] Pascal Fontaine and E. Pascal Gribomont. Combining non-stably infinite, non-first order theories. *Electr. Notes Theor. Comput. Sci.*, 125(3) :37–51, 2005. 21

[92] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9) :490–499, 1960. 60, 65

[93] H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *Proc. 18th IEEE Symposium on Logic in Computer Science,(LICS'03)*, pages 55–64. IEEE Computer Society Press, 2003. 34

[94] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence*, 55(1-2) :101–122, 2009. 22

[95] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 306–320, 2009. 22, 42

[96] Silvio Ghilardi. Model-theoretic methods in combined constraint satisfiability. *J. Autom. Reasoning*, 33(3-4) :221–249, 2004. 21

[97] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Decision procedures for extensions of the theory of arrays. *Ann. Math. Artif. Intell.*, 50(3-4) :231–254, 2007. 34

[98] Robert Givan and David Mcallester. New results on local inference relations. In *In Principles of Knolwedge Representation and Reasoning : Proceedings of the Third International Conference*, pages 403–412. Morgan Kaufman Press, 1992. 34

[99] Georges Gonthier. The four colour theorem : Engineering of a formal proof. In Deepak Kapur, editor, *ASCM*, volume 5081 of *LNCS*, page 333. Springer, 2007. 78

[100] Aarti Gupta and Allan L. Fisher. Parametric circuit representation using inductive boolean functions. In Costas Courcoubetis, editor, *CAV*, volume 697 of *LNCS*, pages 15–28. Springer, 1993. 51

[101] Walter Hamscher, Luca Console, and Johan de Kleer, editors. *Readings in Model-based Diagnosis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. 57

[102] Hartshorne, Weiss, and Burks. *Collected Papers of C.S. Peirce (1930–1958)*. Harvard U. Press. 57

[103] Miki Hermann and Roman Galbavý. Unification of infinite sets of terms schematized by primal grammars. *Theor. Comput. Sci.*, 176(1-2) :111–158, 1997. 50

[104] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. *CoRR*, abs/1605.00723, 2016. 10

[105] Thomas Hillenbrand, Arnim Buch, Roland Vogt, and Bernd Löchner. Waldmeister - high-performance equational deduction. *J. Autom. Reason.*, 18(2) :265–270, 1997. 10

[106] J. Hobbs, M. Stickel, D. Appelt, and P. Martin. Interpretation as abduction. *Artificial Intelligence*, 63 :69–142, 1993. 57

[107] Matthias Horbach and Christoph Weidenbach. Superposition for fixed domains. *ACM Trans. Comput. Logic*, 11(4) :1–35, 2010. 51

[108] Peter Jackson. Computing prime implicates. In *ACM Conference on Computer Science*, pages 65–72, 1992. 60

[109] Swen Jacobs. Incremental instance generation in local reasoning. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2009. 34

[110] Deepak Kapur and Mahadevan Subramaniam. Lemma discovery in automated induction. In *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, pages 538–552, 1996. 49

[111] Deepak Kapur and Hantao Zhang. Rrl : A rewrite rule laboratory. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 768–769. Springer-Verlag, 1988. 48

[112] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Software Eng.*, 23(4) :203–213, 1997. 48

[113] A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9(2) :185–206, 1990. 60

[114] E. Knill, P.T. Cox, and T. Pietrzykowski. Equality and abductive residua for horn clauses. *Theoretical Computer Science*, 120 :1–44, 1992. 60

[115] D. Knuth and P. Bendix. Simple word problems in universal algebra. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970. 10

[116] Konstantin Korovin and Andrei Voronkov. Integrating linear arithmetic into superposition calculus. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2007. 29

[117] S. Lee and D. A. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9 :25–42, 1992. 34

[118] A. Leitsch. *The resolution calculus*. Springer. Texts in Theoretical Computer Science, 1997. 22

[119] Leopold Löwenheim. Über Möglichkeiten im Relativkalkül. *Mathematische Annalen*, 76 :447–470, 1915. 18

[120] Christopher Lynch and Barbara Morawska. Automatic decidability. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 7–, Washington, DC, USA, 2002. IEEE Computer Society. 23

[121] Christopher Lynch and Duc-Khanh Tran. Automatic decidability and combinability revisited. In Frank Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction : Automated Deduction*, Lecture Notes in Computer Science, pages 328–344, Berlin, Heidelberg, 2007. Springer-Verlag. 23

[122] Pierre Marquis. Extending abduction from propositional to first-order logic. In Philippe Jorrand and Jozef Kelemen, editors, *Fundamentals of Artificial Intelligence Research*, volume 535 of *Lecture Notes in Computer Science*, pages 141–155. Springer Berlin / Heidelberg, 1991. 60

[123] A. Matusiewicz, N. Murray, and E. Rosenthal. Prime implicate tries. *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 250–264, 2009. 59

[124] A. Matusiewicz, N. Murray, and E. Rosenthal. Tri-based set operations and selective computation of prime implicates. *Foundations of Intelligent Systems*, pages 203–213, 2011. 59

[125] Marta Cialdea Mayer and Fiora Pirri. First order abduction via tableau and sequent calculi. *Logic Journal of the IGPL*, 1(1) :99–117, 1993. 60

[126] William McCune. OTTER 2.0. In *Proc. of CADE-10*, pages 663–664. Springer, 1990. LNAI 449. 10

[127] William McCune and Larry Wos. Otter - the CADE-13 competition incarnations. *J. Autom. Reasoning*, 18(2) :211–220, 1997. 10, 65

[128] Scott Mcpeak and George C. Necula. Data structure specifications via local equality axioms. In *In CAV*, pages 476–490. Springer, 2005. 34

[129] R. Merton. The theory of rational option pricing. *Bell Journal of Economics and Management Science*, 4 :141–183, 1973. 79

[130] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 272–277. ACM, 1993. 65

[131] Alan Mishchenko. An introduction to zero-suppressed binary decision diagrams. Technical report, in Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, 2001. 60

[132] Lode Missiaen, Maurice Bruynooghe, and Marc Denecker. Chica, an abductive planning system based on event calculus. *J. Log. Comput.*, 5(5) :579–602, 1995. 57

[133] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, 2001. 9

[134] Hidetomo Nabeshima, Koji Iwanuma, and Katsumi Inoue. Solar : A consequence finding system for advanced reasoning. In Marta Cialdea Mayer and Fiora Pirri, editors, *TABLEAUX*, volume 2796 of *Lecture Notes in Computer Science*, pages 257–263. Springer, 2003. 60

[135] Fabrice Nahon, Claude Kirchner, Hélène Kirchner, and Paul Brauner. Inductive proof search modulo. *Ann. Math. Artif. Intell.*, 55(1-2) :123–154, 2009.

[136] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1 :245–257, 1979. 21

[137] A. Newell, J. C. Shaw, and H. A. Simon. Empirical explorations of the logic theory machine : A case study in heuristic. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference : Techniques for Reliability*, IRE-AIEE-ACM '57 (Western), pages 218–230. ACM, 1957. 9

[138] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4) :557–580, 2007. 24

[139] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories : From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *J. ACM*, 53(6) :937–977, 2006. 19

[140] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001. 10, 22

[141] Hans Ohlbach and Jana Koehler. Modal logics, description logics and arithmetic reasoning. *Artificial Intelligence*, 109 :1–31, 1999. 34

[142] Lawrence C. Paulson. Isabelle : The next seven hundred theorem provers. In Ewing L. Lusk and Ross A. Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*, volume 310 of *Lecture Notes in Computer Science*, pages 772–773. Springer, 1988. 78, 79

[143] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, pages 1–11, 2010. 78

[144] D. A. Plaisted and Y. Zhu. Ordered semantic hyperlinking. *Journal of Automated Reasoning*, 25(3) :167–217, October 2000. 34

[145] David Poole. Representing knowledge for logic-based diagnosis. In *FGCS*, pages 1282–1290, 1988. 57

[146] David Poole. A methodology for using a default and abductive reasoning system. *Int. J. Intell. Syst.*, 5(5) :521–548, 1990. 57

[147] Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Comptes Rendus du premier congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929. 18

[148] Alessandro Previti, Alexey Ignatiev, António Morgado, and João Marques-Silva. Prime compilation of non-clausal formulae. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1980–1988, 2015. 60, 72

[149] WV Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9) :627–631, 1955. 59

[150] Franck Ramsay. On a problem in formal logic. *Proc. London Math. Soc.*, 30 :264–286, 1930. 18

[151] Silvio Ranise and David Déharbe. Applying light-weight theorem proving to debugging and verifying pointer programs. *Electr. Notes Theor. Comput. Sci.*, 86(1) :105–119, 2003. 29

[152] R Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1) :57–95, April 1987. 57

[153] John C. Reynolds. Separation logic : A logic for shared mutable data structures. *Logic in Computer Science, Symposium on*, 0 :55, 2002. 77

[154] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In D. Michie and R. Meltzer, editors, *Machine Intelligence*, volume 4, pages 135–150. Edinburg U. Press, 1969. 10

[155] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12 :23–41, 1965. 10

[156] G. Salzer. On the relationship between cycle unification and the unification of infinite sets of terms. In W. Snyder, editor, *UNIF'93*, Boston, (MA,USA), June 1993. 50

[157] S. Schulz. The E Equational Theorem Prover. `http://www4.informatik.tu-muenchen.de/~schulz/WORK/eprover.html`. 10

[158] Yanis Sellami. A DPLL-Based Approach to Prime Implicate Generation. Master's thesis, Grenoble INP, 2016. 76

[159] Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1) :1–12, 1984. 21

[160] L. Simon and A. Del Val. Efficient consequence finding. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 359–370, 2001. 60

[161] Viorica Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In Robert Nieuwenhuis, editor, *CADE*, volume 3632 of *LNCS*, pages 219–234. Springer, 2005. 34

[162] Ryan Stansifer. Presburger"s article on integer arithmetic : Remarks and translation. Technical report, Ithaca, NY, USA, 1984. 18

[163] T. Strzemecki. Polynomial-time algorithms for generation of prime implicants. *Journal of Complexity*, 8(1) :37–63, 1992. 59

[164] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *LICS*, pages 29–37, 2001. 33

[165] The Coq Development Team. *The Coq Proof Assistant Reference Manual V7.1*. INRIA-Rocquencourt, CNRS-ENS Lyon (France), October 2001. `http://coq.inria.fr/doc/main.html`. 79

[166] Cesare Tinelli and Mehdi Harandi. A new correctness proof of the Nelson-Oppen combination procedure. In *Frontiers of Combining Systems, volume 3 of Applied Logic Series*, pages 103–120. Kluwer Academic Publishers, 1996. 41

[167] Cesare Tinelli and Christophe Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theor. Comput. Sci.*, 290(1) :291–353, 2003. 21

[168] Cesare Tinelli and Calogero G. Zarba. Combining nonstably infinite theories. *J. Autom. Reasoning*, 34(3) :209–238, 2005. 41

[169] P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. *Electronic Computers, IEEE Transactions on*, 4 :446–456, 1967. 60

[170] Sophie Tourret. *Abduction in first order logic with equality. (Prime implicate generation in equational logic)*. PhD thesis, Grenoble Alpes University, France, 2016. 61, 67

[171] Duc-Khanh Tran, Christophe Ringeissen, Silvio Ranise, and Hélène Kirchner. Combination of convex theories : Modularity, deduction completeness, and explanation. *J. Symb. Comput.*, 45(2) :261–286, 2010. 23, 61

[172] A. Voronkov. The anatomy of vampire : Implementing bottom-up procedures with code trees. *Journal of Atomated Reasoning*, 15 (2) :237–265, January 1995. 10

[173] Uwe Waldmann and Virgile Prevosto. SPASS+T. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *Proc. ESCoR Workshop, FLoc 2006*, volume 192 of *CEUR Workshop Proceedings*, pages 18–33, 2006. 29

[174] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topic. System description : SPASS version 1.0.0. In *Proceedings of the 16th Conference on Automated Deduction (CADE-16)*, pages 378–382. Springer LNCS 1632, 2001. 10

[175] Makarius Wenzel. Isabelle/jedit - A prover IDE within the PIDE framework. In *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, pages 468–471, 2012. 78

[176] Markus Wenzel and Lawrence C. Paulson. Isabelle/isar. In *The Seventeen Provers of the World, Foreword by Dana S. Scott*, pages 41–49, 2006. 78

[177] Claus-Peter Wirth. Descente infinie + deduction. *Logic Journal of the IGPL*, 12(1) :1–96, 2004. 50

[178] L. Wos, G. Robinson, and D. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM*, 12 :536–541, 1965. 10

[179] Calogero Zarba. Combining sets with cardinals. *J. Autom. Reasoning*, 34(1) :1–29, 2005. 21

[180] Lintao Zhang, Conor Madigan, Matthew Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In Rolf Ernst, editor, *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, pages 279–285, 2001. 9, 19